Priority-Based Scheduling and Evaluation of Precedence Graphs with Communication Times

by

Adel Mohammed Adel Al-Massarani

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

August, 1993

Order Number 1360422

Priority-based scheduling and evaluation of precedence graphs with communication times

Al-Massarani, Adel Mohammed Adel, M.S.

King Fahd University of Petroleum and Minerals (Saudi Arabia), 1993

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

PRIORITY-BASED SCHEDULING AND EVALUATION OF PRECEDENCE GRAPHS WITH COMMUNICATION TIMES

BY

Adel Mohammed Adel Al-Massarani

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

AUGUST 1993

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by **Adel Mohammed Al-Massarani** under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE.**

Thesis Committee

Dr. Habib Youssef
Thesis Advisor

Dr. Sadiq M. Sait
Member

Dr. Subbarao Ghanta
Member

Dr. Samir H. Abdul-Jauwad
Department Chairman

Dr. Ala Al Rabeh
Dean College of Graduate Studies

Date

Abstract

Name: Adel Mohammed Al-Massarani.

Title: Priority-Based Scheduling and Evaluation of Precedence Graphs with Communication Times.

Major Field: Computer Engineering.

Date of Degree: August 1993.

The design of distributed schemes requires partitioning a computation into smaller modules and scheduling it over a number of processors that communicate by exchanging packets of messages. The computation is represented by a directed acyclic graph. The objective function is to minimize the overall computation time. Due to the dynamic nature of the inter-tasks communication, an accurate global estimation of their effect is hard to find. As a result, local scheduling heuristic are used. In this work, new global priority based scheduling algorithms are presented, based on the evaluation of tasks finish times in the reverse graph. Extensive testing is carried out to evaluate the performance of these heuristics. Analysis of the results shows the superiority of these heuristics over reported ones as shorter finish times can be achieved with a stable topology-independant performance. Finally, a new low-cost general iterative improvement technique is presented. Experimental testing of this technique is conducted with an analysis of the process behavior.

Master of Science Degree

King Fahd University of Petroleum and Minerals

Dhahran, Saudi Arabia

August, 1993

خلاصة الرسالة

اسم الطالب: عادل محمد عادل المعصراني.

عنوان الرسالة: جدولة و تقويم المخططات البيانية ذات الأولوية والاتصالات، على أساس الأفضلية.

التخصص: هندسة الحاسب الآلي.

تاريخ الدرجة: أغسطس/ ١٩٩٣م.

ان تصميم الانظمة الموزعة يحتاج الى تجزئة العمليات الحسابية الى أقسام أصغر يمكن جدولتها على عدد من المعالجات التي تتصل فيما بينها عن طريق حزم معلوماتية. أما العمليات الحسابية فتمثل بمخطط بياني موجه لاحلقي. الهدف من الجدولة هو تقليل الوقت اللازم لاتمام العمليات الحسابية. نتيجة للطبيعة المتغيرة للاتصالات بين المهمات، فانه يصعب الوصول الى تقدير شامل لتأثيرها الكلي. ولهذا السبب يتم الاعتماد على طرق الجدولة المحلية.

هذا البحث يقترح أساليب وصيغ خوارزمية جديدة للجدولة على أساس الأفضلية العامة، وتعتمد هذه الأفضلية على تقويم زمن انتهاء المهمات في المخطط البياني العكسي. وقد تم القيام باختبارت مكثفة لتقويم أداء هذه الأساليب، أظهر تحليل النتائج أفضلية الصيغ الجديدة على تلك الموجودة حاليا كونها تؤدي الى جدولة أقصر زمنيا كما أنها تتمتع بأداء ثابت مستقل عن طبيعة شبكة المعالجات. وفي النهاية، يقترح هذا البحث تقنية تكرارية جديدة عامة ذات تكلفة منخفضة لتحسين أداء صيغ الجدولة. وقد تم اجراء اختبارات عديدة لهذه التقنية لتحليل وتقويم أدائها.

درجة الماجستير في العلوم

جامعة الملك فهد للبترول و المعادن

الظهران، المملكة العربية السعودية

أغسطس،١٩٩٣ م

Dedicated to

My Parents

and

Family

Acknowledgment

First, I wish to thank Prof. Mayez Al-Mouhammed with whom most of this work was done. His guidance and help were invaluable. I wish to thank my thesis committee chairman Prof. Habib Youssef for his support and valuable advises, co-chairman Prof. Sadiq Sait for his unlimited help and encouragement and committee member Prof. Subbarao Ghanta for his helpful comments.

I also wish to thank Prof. Mohammed Benten, Dean college of Computer Science and Engineering for giving me full access to the computing facilities at the college. Thanks also to the faculty of the Computer Engineering Department; Department Chairman, Dr. Samir Abdul-Jauwad for their constant encouragement and help.

Thanks to the graduate students of the department of Computer Engineering, specially my dear colleagues Hazem Abu-Saleh and Essam Hubi. Their help and gratitude made the effort and work enjoyable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The introduction of parallel systems, architectures and techniques by early 60's was a landmark in the development of computers. At that time, the available technology at the hardware and software level used in building uniprocessor systems was far behind the demanding computational requirements. The technological barriers severely slowed down the expansion of single CPU speed on the time access. Naturally, the other dimension was to be investigated. By increasing the number of computational resources working simultaneously on the same set of tasks, faster machines could be built, more tasks could be executed within the same interval and different applications could simultaneously exploit and share these common resources. This system of combined processors along with a whole new science created to govern, design and successfully operate it, was called Parallel Processing.

The extent of parallel processing architectures ranges from a simple basic system of interleaved CPU and I/O processor operations, to huge complex systems in terms of number of processors, interconnection network and operating system.

The main concept on which these systems are based upon can be defined as follows. "Parallel Processing is An efficient form of information processing which emphasizes the exploitation of concurrent events in the computing process" [13]. The definition, as it implies, can describe uniprocessor systems that use parallel processing techniques as well as multiprocessing systems. Such techniques range in their level of abstraction from operating system level (as in the use of multiprogramming) to architectural level such as pipeline computers. Parallel processing systems which exploit several CPU's and resources can be classified into two major configurations; Array processors and Multiprocessor systems. Array processors are synchronous multiprocessor systems with SIMD architecture (Single Instruction stream, Multiple Data stream). In general, they are considered quite simpler than Multiprocessor systems which are asynchronous and based on MIMD architecture (Multiple Instruction stream, Multiple Data streams). MIMD systems are more "parallel" in a sense that they allow the decoding and execution of various instruction streams of different tasks rather than decoding one single instruction stream at a time and applying it to different data arrays. The degree of complexity and the nature of both of these architectures is mainly determined by the type of application and environment in which they operate. Array processors are mainly used for processing vector operations, matrix arithmetic, Fast Fourier Transforms (FFT) and other numerical processing applications. MIMD processors are suited for more general applications and time-sharing environments.

The performance of a parallel processing system depends on three major factors; Processors architecture, Processors topology (including the interconnection network) and the operating system. Studies have shown that a high degree of

parallelism can be achieved by using a small set of fast processors rather than a large set of slow ones [13]. Moreover, homogeneous systems, that use an identical set of processors, are more performance-stable, easy to operate and efficient since they require less complex scheduling techniques which are easier to optimize. In general, these systems are more feasible and practical in terms of extendibility because of their scalable architecture. The third performance-determinant factor, is the operating system. The operating system performs the task of distributing the computation over the processing elements. In other words, scheduling. Therefore, the efficiency of the scheduling process can, in fact, determines the efficiency of the whole system. The variety of parallel processing systems and the different criteria measures used in evaluating their performance, results on a variety of scheduling techniques and heuristics.

This thesis presents a new scheduling heuristic for the general MIMD parallel processing model with communication. A new general optimization technique for global scheduling heuristics is also presented. Extensive empirical testing of several variations of this heuristic along with the best reported heuristic (for this model) have been conducted. The rest of the text is divided into five chapters. Chapter 2 introduces the parallel system model and definition along with the terminology. Chapter 3 is a literature review of reported work. The new scheduling heuristic is introduced in chapter 4. The chapter also summarizes the results of the experimental testing. Chapter 5 is on the new scheduling optimization technique along with the testing results. Chapter 6 concludes the thesis.

Chapter 2

Model and Definition

2.1 Objective

Given a computation and a multiprocessor system, the scheduling problem is to map the computation onto the multiprocessor system such that a given objective function is minimized or maximized. In this work, the objective is to minimize the completion time of a partially ordered set of non-preemptive tasks on a set of identical processors. The model presented here is assumed to be deterministic, i.e., all the information governing the scheduling decision is assumed to be known in advance.

2.2 System Model

The scheduling model is described by considering the resources, a task system, sequencing constraints and performance measure. Two models are considered here. The first model, which is assumed in this thesis, takes in consideration an

arbitrary number of messages between any two tasks. Since communication exists between processors, the effect of the processors topology and connectivity arises, which is also considered as well. However, in this thesis we assume contention free media, i.e., we do not consider the issues related to routing and queuing. The second model assumes no communication or message passing between the tasks of the graph and thus no communication between processors. We present this model here to clarify any reference made to it thenceforward.

2.2.1 Contention-free system

A contention free media is one in which the communication channels between processors have sufficient capacity to service all transmissions without a significant delay (due to contention). The contention usually occurs when two or more tasks running in parallel, in a multiprocessor system, send messages through one or more common channels. The effect is to delay the arrival of messages to their destination thus delaying the starting time of their successor tasks [14].

2.2.2 System Definition

A set $\Gamma(T_1, \ldots, T_m)$ of m tasks along with their precedence constraints and communication costs are to be scheduled on a system of n identical processors so that the overall execution time is minimum. The computation Γ can be modeled using a directed acyclic graph (DAG) in which each edge represents the precedence relationship between two nodes or tasks, having each $\mu(T)$ as execution time. With each edge is associated a positive integer $c(T, T\prime)$ which represents the number of messages sent from a task T, upon completion, to its immediate successor

$T\prime$. A general precedence-constrained computation is therefore represented by the quadruple $G(\Gamma, \rightarrow, \mu, c)$ in which arbitrary computation cost $\mu(T)$ and communication cost $c(T, T\prime)$ can be used [13]. The time to transfer $c(T, T\prime)$ messages depends to a large extent on the communication media and the multiprocessor connectivity. The time to transfer a unit of message between processor p and processor $p\prime$ is denoted by $r(p, p\prime)$. In the event the immediate tasks T and $T\prime$ are assigned to run on the same processor ($p(T) = p(T\prime)$) then the communication cost is nil, i.e. $r(p, p\prime) = 0$. When tasks T and $T\prime$ are assigned to different processors, the time to transfer $c(T, T\prime)$ messages is a non-negative time ($r(p, p\prime).c(T, T\prime)$) (assuming contention free media). The multiprocessor model is denoted by $S(P, R)$, where P is a set of identical processors and R is a set of routing times between the processors. By varying the weight $r(p, p\prime)$, the system $S(P, R)$ can be used to model loosely-coupled and tightly-coupled message passing systems including fully and partially connected architectures. The scheduling problem, therefore, consists of mapping the computation $G(\Gamma, \rightarrow, \mu, c)$ onto the system $S(P, R)$ so as to minimize the overall finish time. Figure 2.1 shows an example of a task DAG (a) and the system model (b).

The second model which assumes no communication costs is denoted by $G(\Gamma, \rightarrow, \mu)$. The resources consist of a set $P = \{P1, .., Pn\}$ of processors with no communication in between. In general, the first model can be considered as a superset of the second model. Therefore, the $G(\Gamma, \rightarrow, \mu, c)$ is a special case of the general model where all the $r(p, p\prime)$ values are equal to zero.

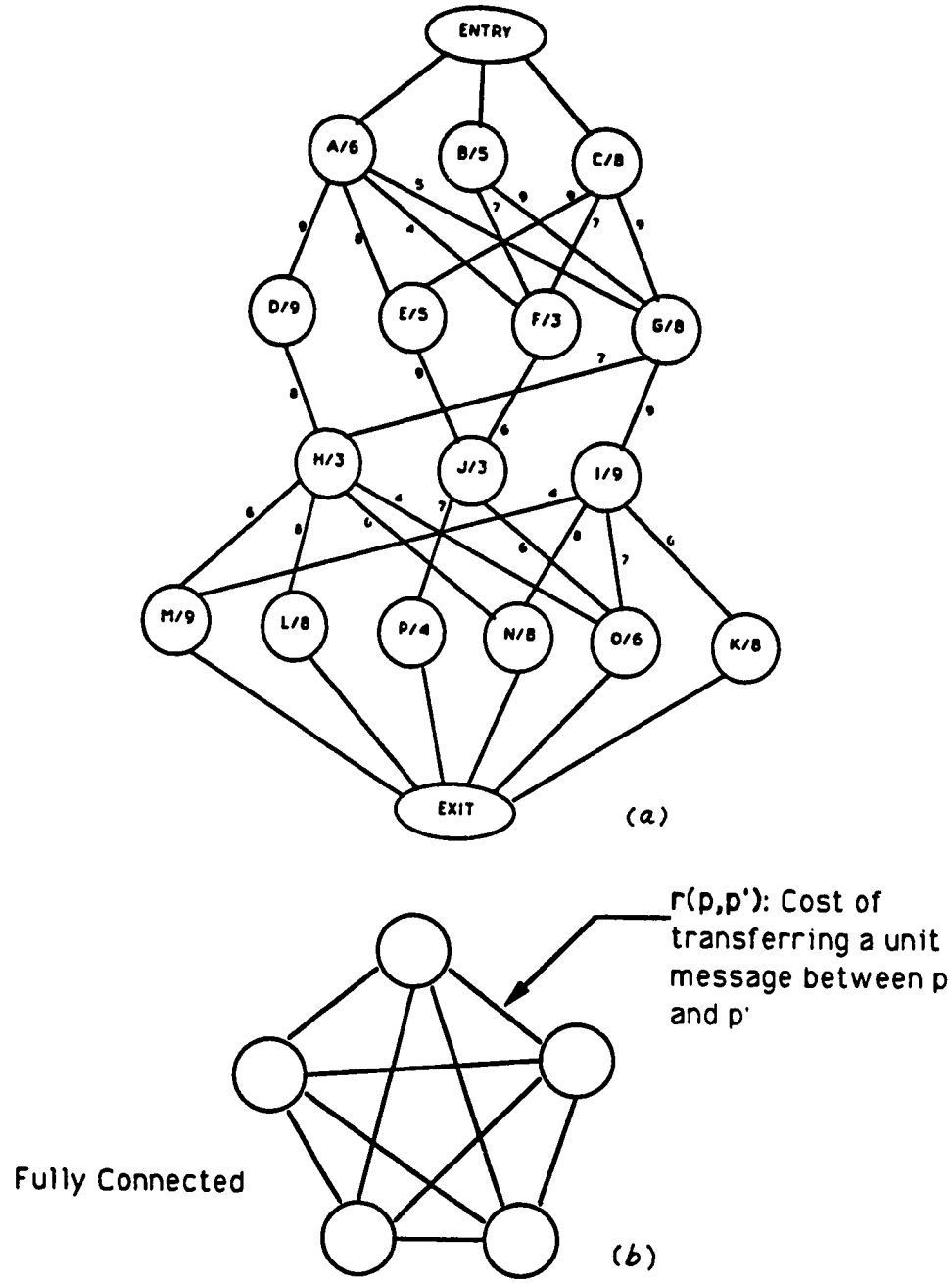

Figure 2.1: Example of: (a) Precedence Graph with Communication Costs (b) Processors Topology.

2.2.3 Preemption Consideration

The scheduling problem presented in the thesis falls into the class of non- preemptive static scheduling. With non-preemptive constraint, a task cannot be interrupted once it has begun execution, and thus, must be allowed to run to its completion. Preemptive scheduling allows a task to be "sliced" into several blocks that can be executed at spanned times on different processors [2]. Preemptive scheduling is not, in general, a practical choice when considering parallel processing systems with message passing and processors inter-communications. Task preemption tends to increase the overall number of messages in the system since each time a preempted task resumes its execution on a different processor, it needs to transfer certain intermediate results and messages from the last run.

2.2.4 Static and Dynamic Scheduling

In static scheduling, the complete precedence-constraints graph is known in advance. The advantages of static scheduling lies in its ability to find more global near-optimum schedules. This is because most static scheduling algorithms exploit the graph global characteristics like critical paths and nodes properties such as the node level, co-level and latest starting time. The major disadvantages of static scheduling is its inadequacy to handle branches and dynamic loops that are determined and unfolded during the program run-time [14]. Dynamic scheduling is more suited for such structures since tasks following branch decisions are scheduled as soon as they are made and loops are handled while unfolded. The major disadvantages of dynamic scheduling are the lack of global measures and the run-time overhead incurred by the on-line scheduling and overhead-processing.

2.2.5 Task Allocation and Scheduling

It's worth mentioning that task allocation and task scheduling are not quite the same. In task allocation, the objective is to minimize the communication delay between processors and to balance the load among them. The nature of the problem is close to the general allocation and placement. Kruatrachue [14] showed that task allocation is not sufficient to obtain minimum schedule finish time since the order of tasks assignment on certain processors and the predecessor-successor tasks alignment is not investigated. In static scheduling these factors are of prime importance. The load balance between processors is not, in general, targeted in static scheduling and is sacrificed for the sake of shorter schedule length. The load balance plays a more important rule in dynamic scheduling since no information is known in advance about the rest of the task graph. So the scheduler will try to fairly distribute the tasks among processors and will try to accomplish the shortest schedule time at any given instant.

2.3 Common Terms

In this section we define some classical terms and scheduling terminology. The actual interpretation of some of these terms may vary according to the model and the type of the graph used.

Length of a Path: The length of a path from a node T_i to a node T_e is the summation of all node execution times along that path from T_i to T_e. Several paths can exist between the nodes T_i and T_e with each path passing by a different set of intermediate nodes. Therefore, the length of these paths would

be different according to the summation values. Using the $G(\Gamma, \rightarrow, \mu, c)$ model, the length of a path is sometimes modified to include the effect of tasks communication. One of the approaches is to add the edge communication delays along the path [14].

Level of a Node: The level of a node T_i with respect to a graph is defined as the length of the longest path from T_i to any exit node. Sometimes, the level of a node is referred to as the Exit path of the node [15]. In mathematical terms, The level l_i of task i is defined to be the longest path length from the task i to an exit node, or

$$l_i = max_k \sum_{j \in \pi_k} t_j \tag{2.1}$$

where π_k represents the kth path from the task i to the exit node, and t_j is the execution time of the task j along that path [16]. As stated before, this classical definition of task level is primarily used in the $G(\Gamma, \rightarrow, \mu)$ model since communication edges are neglected. The more general term used to describe the task level for the $G(\Gamma, \rightarrow, \mu, c)$ is sometimes called the task Latest Starting Time (*lst*). The definition of *lst* has several variations in the literature depending on the heuristic used and thus is defined separately for each case.

Critical Path: The longest path in the graph from any entry node to any exit node.

Co-level of a Node: The length of the longest path from the node to an entry node [3].

Set of Ready-To-Run-Tasks: At any instant during the scheduling, this set holds the schedulable tasks. Schedulable tasks are tasks that have had all their predecessors already assigned on processors and thus eligible for scheduling.

Chapter 3

Literature Review

3.1 Introduction

In this chapter we review the various approaches to solving the scheduling problem. We shall also define the model widely accepted and trace back its evolvement. The extensive research on this problem and the actual statistical and empirical studies performed, have shaped and pointed out the different inherent factors that affect the scheduling problem and the objective function. The enhancement to the basic model of the task graph has gone through several phases to reflect the realistic constraints such as inter-tasks communications, communication delays, task times arbitration, processors connectivity and other system and technology dependent factors.

Although earlier research work in this field was of little practical use for actual systems within the available technology, it determined the basic theory and techniques for the following work. As a matter of fact, early research on scheduling defined the mutual relationship between theoretical work and actual physical sys-

tems. As theoretical work on scheduling theory has affected, in general, the design and implementation of parallel processing systems, the effect of technology and physical constraints on these systems directed the mainstream of research work on scheduling area as well. Most notably, the effect of communication delays, resources and connectivity bound, introduced more complexity on the scheduling problem in general.

The fact that the problem is NP-Complete even when relaxing the constraints and restrictions regarding the communication, network topology and message delays, makes heuristic approaches more feasible and practical in terms of time and resources. Almost all of the research done today on the scheduling problem deals with enhancing the classical heuristic approaches or applying techniques used in other problems such as AI techniques, Branch and Bound, and Linear programming.

3.2 Problem complexity and optimal schedulers

The problem of scheduling a parallel computation onto multiprocessor systems is known to be NP-Complete in its most general form [14]. The difficulty and complexity of the problem varies widely according to several factors such as tasks preemption, precedence arbitration, uniformity of tasks execution times, communication cost, communication to computation ratio, the multiprocessor system topology and characteristics such as the number of processors, processors unity and connectivity [16]. The variation of the problem complexity with respect to these factors is illustrated in Table 3.1.

Nevertheless, a lot of research have been done on restricted forms of the prob-

Problem No.	Number of Processors (m)	Task Processing time (t)	Precedence Constraints	Messages between tasks?	Complexity
1	arbitrary	equal	tree	No	O(n)
2	2	equal	arbitrary	No	$O(n^2)$
3	arbitrary	equal	arbitrary	No	NP-Hard
4	Fixed m >= 2	t_i=1 or 2 for all i	arbitrary	No	NP-Hard
5	arbitrary	arbitrary	arbitrary	No	NP-Hard
6	arbitrary	arbitrary	arbitrary	Yes	NP-Hard

Table 3.1: Complexity of Non-preemptive Scheduling Problems.

lem by imposing constraints on the task graph and the system model. The only known case involving arbitrary precedence structures with no restriction on the task graph for which optimal non-preemptive schedules can be computed in polynomial time requires the assumption of two processors, equal task execution times and no communication [1]. In this case, the algorithm to find the optimal schedule becomes quite simple due to the system minimal requirements. The algorithm uses a simple labeling scheme that assigns to each task in the graph a priority measure. The label identifies the task's importance with respect to the whole graph such that tasks with higher label values have higher priorities. Terminal nodes k are assigned arbitrary values from 1 to k. At the next level, the next task to be labeled $k+1$ is the one with either the lowest successors labels or the least number of successors. After labeling, tasks are scheduled according to their label numbers with higher-label ready-to-run-tasks first.

The assumptions of equal tasks times, two processors scheduling and no inter-tasks communications simplify the calculation of priorities and lead to optimal schedule. The priority in this case depends mostly on the level of successors in the graph and, in the event of a tie, the number of successors. Prastein proved that by taking communication into consideration, the problem of scheduling arbitrary precedence graphs onto two processors becomes NP-Hard [21]. In fact, relaxing any of the previous constraints (for example by allowing arbitrary execution times) makes the scheduling problem NP-Hard.

It has been shown by Coffman that relaxing the assumption of tasks having equal execution times no longer results in optimal schedule [1]. This is because the task priority should also account for its execution time or more generally,

the total execution times of the tasks that lie on the longest path to the exit node. It will be shown later that for the $G(\Gamma, \rightarrow, \mu, c)$ model, task communication should be included in the task priority estimation. However, the problem with the communication is quite more complex since its a dynamic factor that can only be accurately estimated during scheduling and not in advance. This is because the communication cost is waived if both communicating tasks are scheduled on the same processor, or multiplied by the processors communication costs in the case they are scheduled on different processors.

Another classical graph model for which an optimal schedule can be found is when the precedence relation defines a tree [2]. The model also assumes equal tasks execution times, but the number of processors is arbitrary. The algorithm uses a simple strategy that schedules tasks at earliest on any idle processor according to their level value. Whenever a processor becomes free, it is assigned a task, from the set of ready to run tasks, with the highest level value. Although the graph model for the last two optimal cases is simple, the use of task level as a priority measure shows the significance of global information in determining the scheduling decision.

3.3 Scheduling for the $G(\Gamma, \rightarrow, \mu)$ model

By considering the queueing delay, static scheduling has been applied to arbitrary connected networks of processors (Ring, Star, Mesh, etc.,) [14,15,16,17] by using local-priority based scheduling heuristics such as the task with earliest finish time first heuristic [14]. Self-timed scheduling [11] leads the schedule to delay the firing of a task until data dependency is resolved for that task, which removes the need for

polling the data and leads to better synchronization. As the optimum solution is not accessible for all cases, lower bounds [18,19] have been proposed for comparing performance of scheduling heuristics to that predicted by the lower bound which is considered as a reference of the optimum solution. Most static scheduling heuristics have complexity [15] that ranges from $O(n \log n)$ to $O(nm^2)$, where n and m are the number of processors and the number of tasks respectively. Scheduling heuristics have been applied to regular computing such as signal processing [11], LU decomposition, FFTs [14], and robot dynamics [16,17,20].

3.4 Scheduling Techniques and Heuristics

Extensive research was conducted on using branch and bound and simulated annealing [22] as heuristic approaches to mapping and partitioning sub-problems. These approaches either minimize objective functions other than the computation finish time or lack global evaluation because only sub-problems are investigated.

3.4.1 Branch and Bound

Branch and Bound techniques have been used as optimization methods for some existing scheduling heuristics [16]. The optimization method uses branch and bound techniques to search the space of possible schedules. At each step, the number of branches generated for each branching node is equal to the number of combinations to choose from the schedulable tasks. The selection rule that determines at each instance which branches to expand varies according to the heuristic used. Several selection rules have been investigated such as FIFO, LIFO and LLB

(Least Lower Bound) [23,24]. These techniques are practical and produce good results when the task graph is relatively small and the number of processors is moderate. However, when considering large graphs with complex topologies and added communication arcs, the number of possible task-on-processor combinations increases exponentionally at each expanding-branch node. Although some evaluation heuristics are used to limit the combinatorial search space and cut down the number of branches, the high cost of such exhaustive techniques makes it more suitable for small systems or when establishing benchmark tests for measuring scheduling heuristics performance.

3.4.2 Linear Clustering

An approach based on linear clustering has been proposed [25] to iteratively merge the most communicating paths. After multiple refinements, the resulting graph is mapped onto the target multiprocessor using a graph theoretical approach. By considering infinite number of processors, a method [26] based on clustering immediate tasks in order to heuristically minimize the critical path length, has been proposed as an attempt to achieve minimum finish time. When the heuristic minimum critical path length is found, merging operations reduce the number of clusters to match the number of processors.

3.4.3 Load Balancing

Considerable research has been done in this area. The goal of some of reported work was to place tasks that work together, on different processors in order to allow maximum parallelism. On the other hand, other researche had an opposite

goal, which was to find subset of all the tasks in the task graph that work together and place them on the same processor in order to minimize the inter- processor communication [27]. Another goal was to prevent the situation where some processors could be overloaded while others are empty. In general, the research in this area can be classified into two categories which are the graph theoretic models, and the heuristic model.

Graph theoretic models

Graph theoretic model can be used for a system with fixed number of processors, fixed number of tasks, and an accurately estimated average of traffic between any arbitrary pair of tasks.

The problem of allocating the tasks to k processors is reduced to partitioning the task graph into k-subgraphs with minimum number of arcs in between. The arcs between two subgraphs represent then the amount of communication between two CPUs. This problem was largely studied by a number of researchers [27].

Heuristic load balancing

The heuristic load balancing can be used for dynamic tasks scheduling. The idea is to keep track of the load on the processors and to manage the scheduling accordingly. Number of methods were proposed by researchers for estimating the load of the processors [28].

3.4.4 List Scheduling

Scheduling precedence graphs $G(\Gamma, \leftarrow, \mu)$ with no communication costs has been largely studied by using approximation methods. Among the most efficient scheduling heuristics is Graham's list scheduling [1,2,3]. The main strategy of list scheduling heuristics is: no processor is allowed to remain idle if there is some ready to run task that can start on it. Whenever a task is added or deleted from the set of ready to run tasks, these tasks are sorted in a list of descending priority so that when a processor becomes idle, it's assigned the task at the head of the list. The schedulers in this class differ only in the way each scheduler assigns priorities to tasks. Priority assignment results in different schedules because tasks are selected in different order.

Most efficient scheduling heuristics including list scheduling are priority-based algorithms. The task priority could be the earliest or latest starting time of the task. For example, the well known critical path (CP) method (a class of List scheduling heuristics) consists of selecting the ready task whose distance to any exit node is the highest among all the available tasks. In other words, the level of a node or a task is the priority measure when performing the scheduling decision. Other variations of list scheduling heuristics include HLFNET (Highest Level First with No Estimated Times), Random scheduler SCFET (Smallest Co-level first with estimated times) and SCFNET (Smallest Co-level first with no estimated times) [3]. Empirical testing [3] of different scheduling heuristics has shown that priority- based scheduling algorithms exhibit the best performance as the CP method statistically deviates by at most 4.6% from the optimum solution. In fact, Adam has shown that among all priority schedulers, level priority sched-

ulers are the best at getting close to the optimal schedule [3]. The success of these heuristics is due to the use of global information on the computation as the basis for defining the task priority.

3.5 The use of global measures

The evaluation of the shortest distance from the entry node to the completion of a task, i.e. Earliest Completion Time (ECT), and that from the beginning of a task to the exit node, i.e. Latest Starting Time (lst), is the basis for defining the task priority. This measure indicates which task is more critical than others when the objective function is to minimize the computation time. Computation of the ECT and lst is straightforward for the model $G(\Gamma, \rightarrow, \mu)$, i.e. precedence graph with no communication costs. This is because the optimum finish time can be easily evaluated as the longest path from the entry node to the exit node. However, the evaluation of ECT and lst are very difficult to obtain for precedence graphs with communication costs, i.e. $G(\Gamma, \rightarrow, \mu, c)$. The reason is that no information is available to know beforehand how paths in the graph are affected by the communication requirements. This depends on the task assignment to processors and the processor connectivity. Only those tasks that are assigned to the same processor do not need to account for the communication cost. Therefore, the length of a path is difficult to evaluate before the tasks are assigned to processors. The exact values of the ECT and lst depend on determining the optimum finish time of the computation $G(\Gamma, \rightarrow, \mu, c)$ which is an NP-Hard problem.

3.6 Communication Effect

The problem of minimizing the total execution time and communication costs for nonprecedence-constrainted tasks has been investigated in distributed computing systems [7,8]. Precedence-constrained tasks were traditionally studied in scheduling theory for which the early approximation algorithms assume that the inter-task communication cost has negligible effect on the objective function. The assumption on the communication time has been largely based on shared-memory architectures. Due to significant communication overhead, this assumption cannot be justified for message passing systems [9,10]. Therefore, we need to handle the computation with respect to its task-precedence, communication, and the multiprocessor topology [12].

Nevertheless, several attempts have been made to find optimal schedulers for restricted forms of the $G(\Gamma, \rightarrow, \mu, c)$ model. As an example, Anger showed that when there are enough identical processors with identical links (i.e. $r(p, p\prime)$ is the same for all p, $p\prime$) to run all available tasks and when communication delays are no longer than the shortest task processing time, then there is a linear-time optimal algorithm [29]. Furthermore, The algorithm applies only to forests in-trees task graphs on a contention-free media [1]. The restrictions on the task graph, alone, greatly simplify the scheduling problem. The restriction on the communication to computation ratio removes the need to align the chain of tasks with high communication cost in between on the same processor. Furthermore The use of in-forest tree graphs simplifies the selection of joinable tasks and radically cuts down the number of possible predecessor-successor combinations.

[1] Forest-in tree: each mode can only have one successor

3.7 Local priority algorithms

The restrictions imposed by optimal heuristics deviate the $G(\Gamma, \rightarrow, \mu, c)$ model from applicability to real models. On the other hand, the complexity of the problem for the full model invalidates (in terms of time and resources) any optimal or, to certain extent, near optimal scheduling heuristics. To avoid this problem, several researchers [13] have proposed a number of scheduling heuristics based on the use of pure local priority algorithms such as the principle of earliest task first or largest communication first. These approaches generate, with reasonable time and complexity, acceptable solutions when no global information is available. These algorithms are local because tasks priorities do not include global information such as tasks level or *lst* (Latest Starting Time) and thus, priorities are determined dynamically while scheduling. This eliminates the need to pre-process the task graph and therefore simplifies the scheduling process.

One of the best local scheduling heuristics was presented by Hwang [13] and is called "Earliest Task First (ETF)". ETF is event-driven or processor- driven and has a simple greedy strategy : the earliest schedulable task is scheduled first. At each step, the algorithm attempts to schedule a task T on a processor P if the earliest starting time $r(T, P)$ of T on P is the smallest among all the schedulable tasks and all the available processors. The ETF algorithm is purely local because no global information about the tasks is used in the scheduling decision. ETF is a Processor-driven scheduler because task selection (among the set of ready to run tasks) and scheduling decision are done whenever a processor becomes idle ad thus at the end of tasks execution. This approach differs from the task-driven or graph-driven approach for which scheduling evaluation and decision takes place

whenever a task is assigned to a processor and the set-of-ready-to-run tasks is updated.

The ETF algorithm time complexity is $O(nm^2)$, where n and m are the number of processors and the number of tasks, respectively. The relatively high time complexity of ETF, despite the fact that its a local heuristic with no global information processing overhead, compared with time complexities of heuristics for the $G(\Gamma, \rightarrow, \mu)$ model, reflect the effect of communication in increasing the complexity of the scheduling problem.

3.8 Global Measures for the $G(\Gamma, \rightarrow, \mu, c)$ model

Some effort has been made to investigate the use of global information when scheduling and adopting the $G(\Gamma, \rightarrow, \mu, c)$ model. The use of global measures such as the node level (Latest Starting Time) which takes into consideration the communication edges, is not straightforward since the inclusion and exclusion of these edges is a dynamic factor and can only be determined after the scheduling decision and not before. Rewini has used a variation of the Task Level that includes the effect of tasks communication as a priority measure [14]. The Level of a task is modified to add up all the communication edges values along the longest path from the task node to the exit node. As a result, the level of a task is taken to be the path from the task to any exit node accumulating the highest possible summation of tasks times and communication edges. Adding the communication cost directly to the level value introduces inaccuracy in the scheduling decision especially when high communication arcs are waived upon scheduling. This is because communication arcs are assumed to have static rather than dynamic values.

The results obtained by Rewini suggests that the use of such measures increase the chance of better schedule by 1.5 times [14]. However, needless to say, finding efficient list scheduler for the $G(\Gamma, \rightarrow, \mu, c)$ model requires more accurate estimation of task level value which recognizes the dynamic nature of the communication factor.

Chapter 4

GLS Heuristics

4.1 Precedence-Constrained with no Communication costs

For the model $G(\Gamma, \rightarrow, \mu)$ which represents a precedence-constrained computation with no communication costs, the latest starting time or task-level $lst(T)$ of task T can only be evaluated when infinite number of processors are assumed. This means that the accurate values of $lst(T)$ can be easily evaluated if the available processor activity (P_{av}) exceeds the required processor activity (P_{req}), where P_{av} is the number of available processors and P_{req} is the maximum number of tasks that can simultaneously be made ready-to-run at any time. In this condition, the Graham's method for building the priority-list to be used for list scheduling applies. It consists of evaluating the latest starting times for the exit nodes and

propagating up to the entry nodes by applying the following step:

$$lst(T) = \mu(T) + \begin{cases} 0 & \text{if } succ(T) = \phi \\ \max\{lst(T\prime) : T\prime \in succ(T)\} & \text{otherwise} \end{cases} \tag{4.1}$$

where $succ(T)$ denotes the set of successor tasks of T. The Graham's method applies only when $P_{av} \geq P_{req}$. Because no method is known to evaluate the latest starting times when $2 < P_{av} < P_{req}$, the Graham's list scheduling has been applied as a heuristic approach for scheduling precedence graphs for arbitrary number of processors. It has been shown that the Graham's method is near optimum in both the deterministic and stochastic cases because the generated solution statistically deviates by nearly 5% only from the optimum solution.

For the model $G(\Gamma, \rightarrow, \mu)$, the computation of latest starting time is independent of the ordering of the tasks as the computation proceeds because there is no communication cost. Immediate tasks that are allocated to identical or different processors cannot be delayed due to their processor assignments. Therefore, evaluation of the $lst(T)$ times when $P_{req} \leq P_{av}$ is independent of the mapping of tasks to processors. When $P_{av} < P_{req}$ the accurate evaluation of $lst(T)$ implicitly means that the tasks have been assigned to the processors so as to yield optimum overall finish time. As there is no communication, the task assignment results from partitioning the graph into P_{av} sets of sequential tasks such that each set is mapped to an arbitrary processor. Therefore, even when $P_{av} < P_{req}$ the evaluation of $lst(T)$ is only affected by the number of available processors but not of the identity of these processors,i.e. no mapping is required. This leads to the following statement of the latest starting time: "$lst(T)$ is the latest starting time of task $T \in \Gamma$ that does not lead to an increase of the optimum finish time of a precedence constrained

computation $G(\Gamma, \rightarrow, \mu)$ over P_{req} processors".

4.2 Precedence-Constrained Computation with Communication

Consider the problem of scheduling a precedence-constrained computation $G(\Gamma, \rightarrow, \mu, c)$ with communication costs over a message passing multiprocessor $S(P, R)$. Let $T \in \Gamma$ be a task and let $succ(T)$ and $pred(T)$ be the set of immediate successors of task T and the processor to which task T is assigned, respectively. In the event, the immediate successor $T\prime \in succ(T)$ is assigned to run on $p(T)$, then no communication is required because the $c(T, T\prime)$ messages are available within the private resource of processor $p(T)$. Therefore, task $T\prime$ can start running at the completion time ($f(T)$) of T, provided that all the precedence constraints of $T\prime$ are satisfied. In the event $T\prime$ is assigned to a processor $p(T\prime) \neq p(T)$, processor $p(T)$ transfers $c(T, T\prime)$ unit messages from $p(T)$ to $p(T\prime)$ such that the last unit of message reaches $p(T\prime)$ at time $f(T) + c(T, T\prime)r(p(T), p(T\prime))$, where $r(p(T), p(T\prime))$ is the time to transfer a unit of message from $p(T)$ to $p(T\prime)$ when the routing media is contention free. Therefore, the earliest starting time of $T\prime$ with respect to its predecessor T is given by:

$$s(T\prime) = f(T) + \begin{cases} 0 & \text{if } p(T) = p(T\prime) \\ c(T, T\prime).r(p(T), p(T\prime)) & \text{otherwise} \end{cases} \tag{4.2}$$

Evaluation of the latest starting times for optimizing the computation of $G(\Gamma, \rightarrow, \mu, c)$ on a multiprocessor $S(P, R)$ is dependent on the mapping of the tasks to processors. The problem of evaluating the $lst(T)$ times depends on the knowledge

of the optimum finish time for at least the case where $P_{req} \leq P_{av}$. This information is not available because the problem of optimally scheduling $G(\Gamma, \rightarrow, \mu, c)$ over an infinite number of processors has not been shown so far to be tractable. Therefore, heuristic approaches should be investigated for estimating the values of $lst(T)$ in order to enable the design of efficient global priority-based scheduling heuristics. In the following we present a new heuristic evaluation of the latest starting times.

Let T be a task and let $succ(T) = \{T_1, \ldots, T_h\}$ be the set of successors of T that are assigned to processors $p(T_1), \ldots, p(T_h)$, respectively. The latest starting time $lst(T,p)$ of T on processor p is heuristically defined follows:

$$lst(T,p) = \mu(T) + \begin{cases} 0 & if\, succ(T) = \phi \\ \max\{lst(T_i) + c(T,T_i).r(p,p(T_i)) : T_i \in succ(T)\} & \text{otherwise} \end{cases} \tag{4.3}$$

When task T has no successors ($succ(T) = \phi$), the $lst(T)$ time is simply the computation time of T. When T has at least one successor ($succ(T) \neq \phi$) and T is assigned to processor $p(T)$, then $lst(T,p)$ depends on the latest time at which the $c(T,T_i)$ messages are transferred from processor $p(T)$ to processor $p(T_i)$ at a rate of one unit of message per $r(p,p(T_i))$ seconds. As $lst(T,p)$ depends on the routing $r(p,p(T_i))$, then there exists a processor p^* such that T could start at the latest time, i.e. shortest distance from the starting of T to some exit node. This allows heuristically defining the latest starting time associated with task T :

$$lst(T) = lst(T,P^*) = \min_p\{lst(T,p)\} \tag{4.4}$$

The above heuristic evaluation of the $lst(T)$ times is not achievable in the general case because two different tasks may occupy overlapped activity intervals with

respect to the same processor. In other words, the computation of $lst(T)$ may lead the latest activity intervals of T_1 and T_2 to satisfy:

$$[lst(T_1,p), lst(T_1,p)+\mu(T_1)] \cap [lst(T_2,p), lst(T_2,p)+\mu(T_2)] \neq \phi$$

An achievable evaluation should operate by recording the assignment to avoid overlapped activity intervals. In the following we present a heuristic algorithm (lst) to evaluate the achievable values of the $lst(T)$ times.

Algorithm lst evaluates the latest-starting-time of the tasks as they can be achieved by using the best local heuristic. Consider the reverse graph that is obtained by reversing the direction of all the arcs in the original task graph. Heuristic lst performs scheduling of the reverse graph using the earliest-task-first as task-selection criteria over the target multiprocessor $S(P,R)$. The earliest-completion-times $ect(T)$ of the tasks in the reverse graph represent the latest-starting-times in the original graph. The earliest-starting-time of T is defined as the earliest time by which all the communication from the predecessors of T reach some processor p^* :

$$est(T,p^*) = \min_P\{\max_{T' \in D(T)}\{f(T') + c(T',T).r(p(T'),p)\}\} \tag{4.5}$$

where $D(T)$ denotes the set of predecessors of T. Let $f(p)$ be the earliest time processor p becomes free. Therefore, the effective earliest-starting-time of T is the least time at which T can effectively start on some processor p^* by considering the graph requirements and the current free time of all the processors:

$$est(T,p^*) = \min_P\{\max_P\{\max_{T' \in D(T)}\{f(T') + c(T',T).r(p(T'),p)\}, f(p)\}\} \tag{4.6}$$

Algorithm *lst* uses the effective earliest-starting time as task-selection criterion. This notion is used as an approximation method to find the length of the shortest path from starting a task to any exit node. To achieve this objective, *lst* starts with the entry nodes of the reverse graph, determine their *ect* times, and propagate these values down to any task whose predecessors have all been allocated their $ect(T)$ times until the exit nodes are reached.

Algorithm *lst* operates on the reverse graph and uses set B to store the tasks that have already been assigned their $ect(T)$ times and set A to store the tasks that have no predecessor or whose predecessors all belong to B. The functions $est(T,p)$, $ect(T)$, and $f(p)$ denote the earliest-starting-time of T, the earliest- completion-time of T, and the current free time of processor p, respectively. We assume the function $\lambda_d(T)$ is initialized to the number of predecessors of T and is decremented by *lst* every time a predecessor $T\prime$ of T is allocated its $ect(T\prime)$ time. Algorithm *lst* is the following:

ALGORITHM lst

Inputs : Task graph $G(\Gamma, \rightarrow, \mu, c)$ and System S(P,R)

(1) - Obtain the reverse graph by reversing all arc directions.

(2) - $A \leftarrow T : D(T) = \phi, est(T,p) = 0$ for each task $T \in A$ and each processor,
$B \leftarrow \phi, f(p) = 0$ for each processor

(3) - While $|B| < n$ Do
Begin

(3.1) - Select the task $T^* \in A$ and processor p^* that satisfy:

$$est(T^*,p^*) = \min_{T \in A}\{\min_{p}\{est(T,p)\}\}$$

(3.2) - Assign T^*on $p^* : p(T^*) = p^*, ect(T^*) = est(T^*, p^*) + \mu(T^*)$,

$f(p^*) = ect(T^*)$,

Remove T^* from $A : A \leftarrow A - \{T^*\}$,

Append T^* to $B : B \leftarrow B + \{T^*\}$,

For each $T \in A$, update its $est(T, p^*)$:

$$est(T, p^*) = \max\{est(T, p^*), f(p^*)\}$$

(3.3) - Repeat for each task $T \in succ(T^*) : \lambda_d(T) = \lambda_d(T) - 1$,

if $\lambda_d(T) = 0$ Then

Update $A : A \leftarrow A + \{T\}$,

Evaluate the effective $est(T, p)$ for each processor p :

$$est(T, p) = \max\{\max_{T' \in D(T)}\{f(T') + c(T', T).r(p(T'), p)\}, f(p)\}$$

End.

Outputs : List of {ect(T)} of the reverse graph that is identical to the list $\{lst(T)\}$ for the original graph.

The main loop of *lst* is statement 3 which executes n times, since each task is allocated for each run of the body, and there are n. Statement 3.1 executes at most pn times in order to select one task among those of set A and one processor among p processors. Statement 3.2 updates the parameters but its last sub-statement executes n times. The decrementation $\lambda_{d(T)} = \lambda_{d(T)} - 1$ in statement 3.3 executes $O(n^2)$ times but the condition $\lambda_{s(T)} = 0$ occurs only once for each task, then the total number of times to evaluate all needed $est(T, p)$ is pn^2. Therefore, the time complexity of *lst* is $O(pn^2)$.

4.3 Generalized List Scheduling

In this section we present a class of scheduling heuristics that use the global priority-based information in scheduling precedence graphs with communication costs on fully or partially connected multiprocessors. This class of scheduling heuristics results from generalizing Graham's list- scheduling for computations where the inter-task communication aspects and the multiprocessor connectivity should be considered. We call this class of heuristics Generalized List Scheduling (GLS) .

A heuristic that belongs to this class has two steps: 1) evaluate the $lst(T)$ times using the computation model $G(\Gamma, \rightarrow, \mu, c)$, the system $S(P, R)$, and algorithm lst as operator, and 2) scheduling: among all the ready-to-run tasks, select the task with the highest global priority and assign it to the idle processor that can start it at the earliest. The first step is always implemented using algorithm lst and the second step can be achieved using different scheduling strategies that will be presented below.

For the GLS class, a heuristic algorithm can either be processor-driven (PD) or graph-driven (GD). For processor-driven scheduling, the updating of the set of ready-to-run tasks is processor-oriented as it is done at the time a processor completes execution of a task and becomes idle. The successors of these newly completing tasks are only involved in the updating process. This leads PD to track the monotonously increasing sequence of processor idle times. For example, the use of PD with the earliest-task-first leads to the expansion of the task graph rather horizontally by giving equal opportunity to all the paths. By applying this strategy, PD locally attempts to minimize the processor idle times in order to achieve the global objective function. An algorithm that implements the processor-driven approach within the framework of a local scheduling heuristic can be found in [13].

For graph-driven scheduling, the set of ready-to-run tasks is updated following the assignment of each task and only the successors of that task are examined to find out whether some of them are becoming candidates for the next assignment. The set of ready-to-run tasks consists of all the tasks whose predecessors have already been started but not necessarily completed. This approach leads to a depth-first (vertical) expansion of the task graph specially when using the global task-priority to differentiate between critical and non-critical paths. The main strategy of GD is to selectively attempt serializing highly communicating tasks along critical paths on the least costly communicating processors in order to optimize the objective function. The PD approach like the one used in the ETF (Earliest Task First) algorithm [13] relies on time or event management so that it enables the successors of an assigned tasks to be included in the set of ready to run tasks only if there is a chance that they can start earlier than some existing task in the set. Otherwise, existing tasks in the set have to be all scheduled first. The result is horizontal expansion of the graph. We will investigate these two approaches (GD and PD) with respect to their performance.

We define four heuristics that belong to the GLS class. The first heuristic is Graph-Driven/Highest-Level-First (GD/HLF). Selecting tasks according to highest- level-first is identical to highest $lst(T)$ time because $lst(T)$ represents an approximation of the shortest path from starting T to any terminal node including the effects of computation, communication, and multiprocessor topology. The heuristic PD/HLF is similar to GD/HLF but it updates the set of ready-to-run tasks according to the processor-driven approach.

Selecting tasks according to highest-level-first, or highest $lst(T)$, may lead to increasing the processor idle time that precedes the starting of every task. To overcome this effect, the task-level should be affected by the idle time that is created following the starting of the task. For this, we define the heuristic Graph-Driven/Highest-Level-Earliest-Task-First $GD/HLETF$ which gives the highest

priority to the task whose $lst(T) - est(T)$ is the highest, where $est(T)$ is the effective earliest-starting-time (Eq. 6) of T among all the processors. The $est(T)$ is the earliest time at which the precedence constraints and the inter-task communications along the routing media will be satisfied at the starting of T on some processor p that satisfies: $est(T) = est(T, p)$.

Assume T has the highest priority among all the ready-to-run tasks, then

$$lst(T) - est(T) \geq lst(T_i) - est(T_i)$$

for all ready-to-run tasks T_i. In other terms, we have

$$lst(T) - lst(T_i) \geq est(T) - est(T_i)$$

If T and T_i were competing for different processors ($est(T) = est(T, p)$, and

$$est(T_i) = est(T_i, p'))$$

then starting T on processor p will not delay T_i which will compete later for its best suited processor p'. In this case, it does not matter which task is scheduled first. Assume T and T_i were competing for the same processor, then T will be selected only if the difference in levels ($lst(T) - lst(T_i)$) is greater than the amount of idle time ($est(T) - est(T_i)$) that will be saved in the event T_i was scheduled before T. Therefore, task-selection according to highest $lst(T) - est(T)$, i.e. $HLETF$, is an attempt to overcome the difficulty of global priority-based heuristics with respect to processor efficiency. To evaluate this strategy within the processor-driven approach, a heuristic $PD/HLETF$ is defined. Next, we give a graph-driven algorithm for the global priority $HLETF$.

Algorithm $GD/HLETF$ uses set B to store the tasks that have been scheduled on some processors and set A to store the tasks that are currently ready-to-run. Functions $s(T)$, $f(T)$, and $f(p)$ denote the starting time of T, the finishing time of T, and the current free time of processor p, respectively. $D(T)$ and

$succ(T)$ denote the set of predecessors and successors of task T, respectively. The function $\lambda_{d(T)}$ is initially set to the number of predecessors of T and it is decremented by $TD/HLETF$ every time a predecessor $T\prime$ of T is scheduled. Algorithm $GD/HLETF$ is the following:

ALGORITHM GD/HLETF Inputs: Task graph $G(\Gamma, \rightarrow, \mu, c)$, System $S(P, R)$, and list of task priority $\{lst(T)\}$

(1) $A \leftarrow \{T : D(T) = \phi\}, est(T,p) = 0$ for each task $T \in A$ and each processor, $B \leftarrow \phi$, $f(p) = 0$ for each processor.

(2) While $|B| < n$ Do

Begin

(2.1)- Select the task $T^* \in A$ and processor p^* that satisfy:

$$prt(T^*, p^*) = lst(T^*) - est(T^*, p^*) = \max_{T \in A}\{lst(T) - \min_{p}\{est(T,p)\}\}$$

(2.2) Assign T^* to run on p^* : $p(T^*) = p^*, st(T^*) = est(T^*, p^*)$,

$f(p^*) = st(T^*) + \mu(T^*)$,

Remove T^* from $A : A \leftarrow A - \{T^*\}$,

Append T^* to $B : B \leftarrow B + \{T^*\}$,

For each $T \in A$, update its $est(T, p^*)$: $est(T, p^*) = \max\{est(T, p^*), f(p^*)\}$.

(2.3) Repeat for each task $T \in succ(T^*)$: $\lambda_d(T) = \lambda_d(T) - 1$,

If $\lambda_d(T) = 0$ Then

Update $A : A \leftarrow A + \{T\}$,

Repeat for each processor p:

$$est(T,p) = \max\{\max_{T\prime \in D(T)}\{f(T\prime) + c(T\prime, T).r(p(T\prime), p)\}, f(p)\}$$

End.

Outputs: List of $st(T)$ and $p(T)$.

In the following, we explain algorithm $GD/HLETF$ and establish its time complexity. Statement 2 of algorithm $GD/HLETF$ is the main loop that executes n times because one task is scheduled in each run of the body. Statement 2.1 finds task T^* and processor p^* that maximizes the function $lst(T) - est(T,p)$ among all the tasks of A. Statement 2.1 executes np times. Following the scheduling of T^* on processor p^*, Statement 2.2 updates the $est(T,p^*)$ for all the tasks of A by using the new processor free time $f(p^*)$. This statement executes n times. Statement 2.3 evaluates the earliest-starting-time for each newly ready-to-run task by considering the task-precedence and the current processor free time. The condition $\lambda_d(T) = 0$ in statement 2.3 can be achieved only once for each task. Then, to evaluate all needed $est(T,p)$, the total number of times is $O(pn^2)$ that is the time complexity of $GD/HLETF$.

Our main concern is to investigate different methodologies for implementing efficient processor utilization within the framework of global priority-based scheduling. An optimization technique that will be used here is to attempt filling the idle time created following the scheduling of every task by other ready tasks provided that the original task is not delayed. In the event the task T with the highest priority (highest $lst(T) - est(T)$) is going to be preceded by some idle time interval I, the list of ready-to-run tasks is scanned to find the highest-level task that fits interval I without delaying T. If such task $T\prime$ is found, updating the set of ready-to-run tasks with any newly ready successor $T"$ of $T\prime$ cannot cause any delay to T because T is still the most prior task as:

$$lst(T) - est(T) \geq lst(T\prime) - est(T\prime) \geq lst(T") - est(T")$$

Implicitly, this technique attempts filling the idle time I with lower priority tasks until no ready-to-run tasks can fit interval I. Implementing and applying this technique for GD/HLF and $GD/HLETF$ leads to the new heuristics GD/HLF^*

and $GD/HLETF^*$, respectively. The optimization only increases the constant in $O(pn^2)$ which is the time complexity of (GD/HLF) and $(GD/HLETF)$, because the list of ready-to-run tasks is scanned another time to find a task that fits the idle interval.

Other heuristics that will be considered in this work are PD/ETF [13] and GD/ETF which apply the principle of Earliest-Task-First. Algorithm PD/ETF is one of the most known local scheduling heuristics that gives the highest priority to the earliest startable task as defined by Equation 4.5. The earliest-starting-time for PD/ETF accounts only for the precedence constraints and the communication from the predecessors but the effective *est* is not used at the task selection level. Finally we add the heuristic GD/ETF that operates according to the proposed graph-driven approach and uses the effective earliest-starting-time as defined in Equation 4.6. Note that algorithm *lst* uses the heuristic GD/ETF which is applied to the reverse graph.

An algorithm called *Random* is used to show the effect of random task- selection versus deterministic selection. It randomly selects a ready-to-run task and assigns it to the processor that enables its earliest starting time. Since all the studied heuristics start tasks at their earliest-starting-time, this algorithm only introduces randomness at the task-selection level. *Random* is also an indicator for the data-flow concept where tokens can run on any free actor regardless of any priority concept.

An example of precedence-constrained computation is shown on figure 4.1 where the nodes and the arcs indicate the computation and communication times. For simplicity, we consider the scheduling of the graph shown on figure 4.1 onto a 3-processor fully-connected system, i.e. the interprocessor communication cost $r(p, p\prime) = 1$ whenever $p \neq p\prime$ and zero otherwise. The schedule generated by using the local heuristic PD/ETF is shown on figure 4.2-a where each task is followed by the scheduling decision number. For GLS, the task-levels are evaluated using

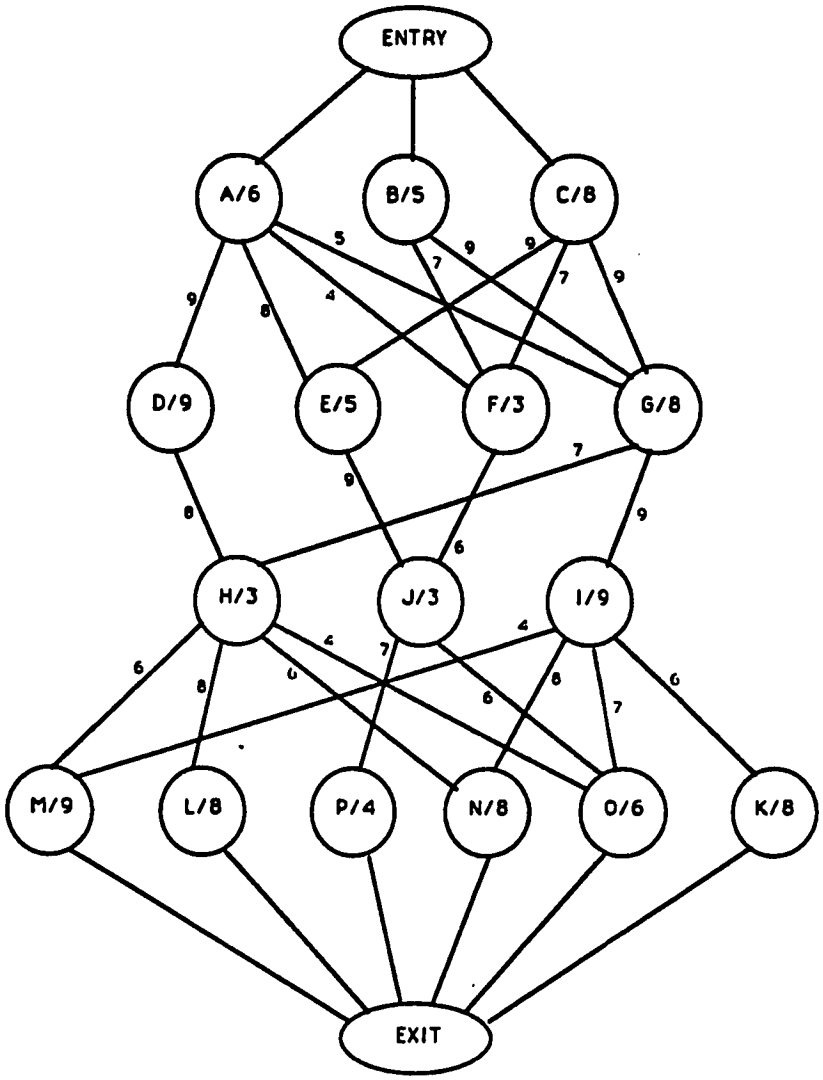

Figure 4.1: A Precedence Graph with Communication Costs.

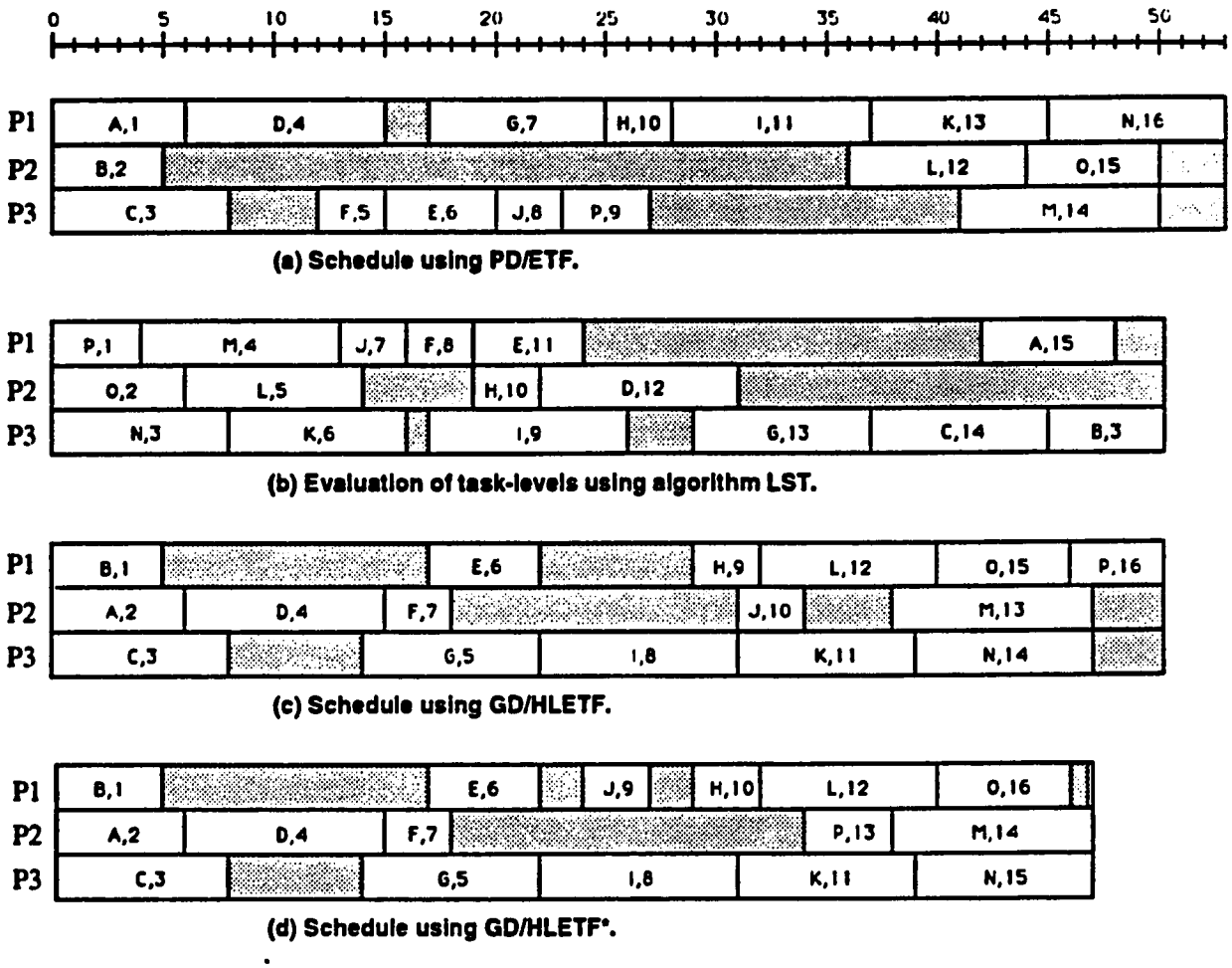

Figure 4.2: Schedules using heuristics (a) PD/ETF, (b) LST, (c) $GD/HLETF$, and (d) $GD/HLETF^*$.

algorithm *lst* and the finish times of the tasks in the reverse-graph are the $lst(T)$ times. For each scheduling decision of $GD/HLETF$, figure 4.3 shows the ready-to-run tasks that are sorted in the decreasing order of $lst(T) - est(T)$ and the selected task. The corresponding schedule is shown on figure 4.2-c. The generated schedule by using $GD/HLETF^*$ is shown on figure 4.2-d. Tasks J and P are scheduled by $GD/HLETF^*$ in the idle times that precede tasks H and M in figure 4.2-c, respectively. Using $GD/HLETF^*$, task J is scheduled (decision 9 in figure 4.2-d) in the idle time that precedes H and, as a result, task P is executed (decision 13 in figure 4.2-d) in the idle time that precedes M. These decisions cannot delay the most prior tasks H and M.

4.4 Graph Generation and Empirical Testing

To evaluate the performance of the obtained scheduling, a random graph generator (RGG) is implemented and used for empirical testing of the proposed heuristics. This will allow the generation of computation graphs with various characteristics, the application of the above heuristics as operators to the input graph, and the collection of results. For each generated graph, the number of tasks ranges from 50 to 200 and the task computation time ranges from 10 to 190 units. The average communication costs, the average number of levels, and the number of processors and their connectivity are controlled using the following parameters:

1. The ratio (α) of average communication carried by each edge (C_{arc}) to the average task computation time (μ_T), i.e. $\alpha = C_{arc}/\mu_T$. Thirteen values ($\alpha = 0, 0.05, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 1, 1.5, 2, 2.5, and 3$) are studied.

2. An estimate of the degree of concurrency (β), that is, the average number of tasks in the graph (N_T) over the product of the average number of levels (N_L) by the number of processors (p), i.e. $\beta = N_T/(N_L p)$. Note that the

S.D.	Ready-to-Run T(lst,est,P)	Schedule
1	B(50,0,1), A(48,0,1), C(45,0,1).	B on P1
2	A(48,0,2), C(45,0,2).	A on P2
3	C(45,0,3), D(31,6,2).	C on P3
4	D(31,6,2), G(37,14,3), E(24,14,3), F(19,12,3).	D on P2
5	G(37,14,3), E(24,14,3), F(19,12,3).	G on P3
6	E(24,17,1), F(19,15,1), I(26,22,3), H(22,23,3).	E on P1
7	F(19,15,2), I(26,22,3), H(22,23,3).	F on P2
8	I(26,22,3), H(22,23,3), J(16,24,1).	I on P3
9	H(22,29,1), J(16,24,1), K(16,31,3).	H on P1
10	J(16,31,2), K(16,31,3), L(14,32,1), M(13,35,1), N(8,38,3).	J on P2
11	K(16,31,3), L(14,32,1), M(13,35,1), N(8,38,3), P(4,34,2), O(6,38,2).	K on P3
12	L(14,32,1), M(13,35,1), P(4,34,2), N(8,39,1), O(6,38,2).	L on P1
13	M(13,38,2), P(4,34,2), N(8,39,2), O(6,38,2).	M on P2
14	N(8,39,3), O(6,40,1), P(4,41,1).	N on P3
15	O(6,40,1), P(4,41,1).	O on P1
16	P(4,46,1).	P on P1

Figure 4.3: Tasks selection throughout the scheduling process.

term N_T/N_L is an estimate measure of the graph-concurrency or average number of tasks per level. Parameter *β(0.5, 1, 2, 2.5, 3, and 4)* can be used as an indicator of the average number of tasks that compete to run on each processor within the set of ready-to-run tasks, i.e. the ratio of the required activity over the available activity.

3. The degree of graph-connectivity (γ) that controls the topology of the computation. For regularly connected graphs, the tasks are uniformly distributed over the levels and the communication arcs only connect successive levels of the graph. For irregularly connected graphs, the communication arcs arbitrary connect tasks of different levels while the graph is constrained to be directed and acyclic.

4. The topology of the multiprocessor (δ) that is the Fully- Connected (FC), the Hypercube (HC), and the Ring (RG). The effect of contentions within the interconnection network are not studied in this thesis.

For each instance of α, β, γ and δ, the random graph generator (RGG) uses the uniform distribution to generate 500 graphs that are used as inputs by the previously defined scheduling heuristics. The performance of each heuristic along the graphs of each instance is saved on a file so that absolute and relative achievement, and variance analysis can be performed using the data. Because of the extensive testing and the use of radically different scheduling heuristics, the shortest finish time (ω_{best}) that is achieved by the heuristics for each generated task-graph is considered as a reference of the optimum solution. Due to task-selection and the way the graph is expanded, the performance of the heuristics can be influenced by different factors. The testing will help us identify any algorithm-problem dependencies.

4.5 Performance Analysis

The objective function of the scheduling heuristics is the finish time. The performance analysis compares the relative merits of the local heuristics to the proposed global heuristics.

Our main objective is to study the relative merit of local and global scheduling heuristics. The testing shows that the graph connectivity has a marginal effect on the relative achievement of each heuristic with respect to the others. Decreasing the degree of graph connectivity has been carried out to find its impact on the relative performance of the heuristics. Irregularly connected task graphs having 40%, 30%, and 20% of the total number of communication arcs that link tasks at arbitrary levels affect the relative performance of the studied heuristics by at most 2% within the condition stated in the previous section. The primary results of the testing show that the most important factors that affect the relative performance are: the ratio α of communication to computation, the degree of concurrency β, and the multiprocessor topology. In the following sub-sections we analyze the obtained results and outline the major factors that lead to significant deviation in achieving the objective function.

4.5.1 Deterministic versus Random Task-Selection

Figure 4.4 shows the percentage deviation of the average finish time[1] from ω_{best} that is achieved by *Random* for the *FC* topology. This heuristic only introduces randomness at the task selection level because every selected task is assigned to start on the processor that enables its earliest-starting-time. For low values of

[1] If n graphs are generated, then the average finish time =

$$\frac{\sum_{i=1}^{n} f(G_i)}{n}$$

where $f(G_i)$ is the finish time of a task graph G.

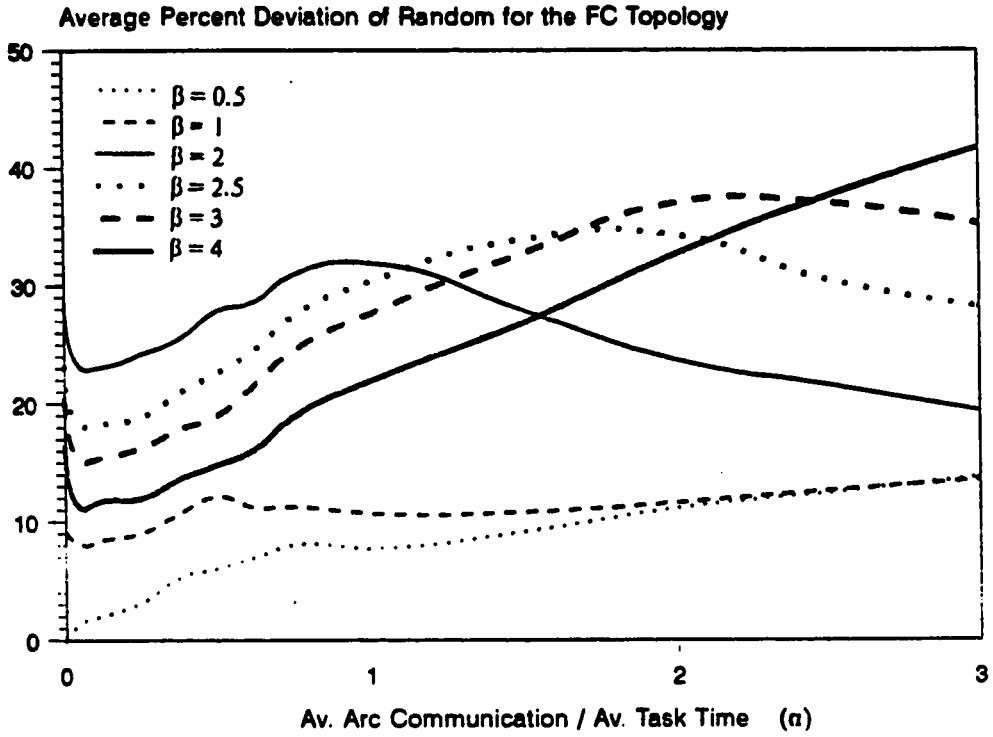

Figure 4.4: The relative performance of Random Heuristic for the FC architecture.

gpaph- parallelism, i.e. β= *0.5 and 1*, the size of ready-to-run tasks is reduced and consequently *Random* has moderate relative deviation from ω_{best} compared with its deviation for higher values of β. This moderate deviatimn is due to the fact that task selection has a slight effect on global performance because the"number of tasks that become ready-to-run is small anyway compared to the number of available processors, making the effect of priority selection insignificant. On the other hand, higher values of β (≥ 2) lead *Random* to substantially deviate from ω_{best} which shous that in this range of β deterministic task-selection becomes an important factor to aahieve the objective function. Analysis of *Random* for the HC and RG topologies supports vhe above observations.

4.5.2 GLS Scheduling with the Processor-Driven Approach

Heuristics PD/HLF and $PD/HLETF$ give low average deviation, i.e. 4% from ω_{best}, in the range of low communication ($0 \leq \alpha < 0.5$) and for all studied values of graph-parallelism β. However, these heuristics significantly deviate from ω_{best} in the range of medium to high communication ($0.5 \leq \alpha \leq 3$). For the FC topology and in the range of high communication, the peak deviation of $PD/HLETF$ and PD/HLF are 9% and 60% from ω_{best}, respectively. This effect is also shown in figure 4.5 for heuristic $PD/HLETF$ with the HC topology.

Heuristic PD/HLF is equivalent to the traditional $CP/HLFET$ except that the tasks are delayed until completion of the communication. The strong deviation of PD/HLF from ω_{best} indicates that the use of traditional list-scheduling, that is near-optimum for the model $G(\Gamma, \rightarrow, \mu)$, leads to inefficient solutions for precedence- constrained computation with communication and arbitrary multiprocessor topology.

The reason for these strong deviations is that the processor-driven approach is suitable for heuristics that locally attempt minimizing the processor idle times in

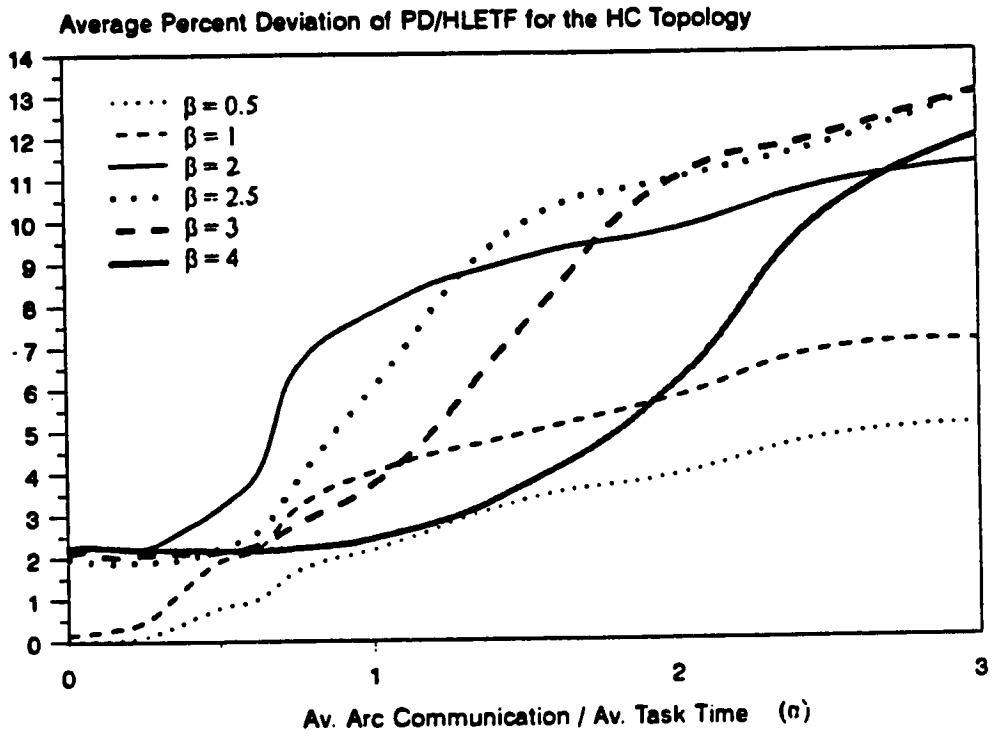

Figure 4.5: The relative performance of $PD/HLETF$ for the HC topology.

order to achieve shorter finish time. The PD approach requires efficient tracking of the idle time rather than selecting tasks according to global priority. The use of PD/HLF and at a lower level $PD/HLETF$ lead to inefficient utilization of the processor idle times where the communication requirements are significant. The use of $est(T)$, as in the case of $PD/HLETF$, has overcome most of PD/HLF deficiency because of better processor utilization. This effect is also emphasized for the HC (12%) and RG (15%) topologies because of increasing communication penalty between the processors compared to that of the Fully-connected topology. As a result, their acceptable performance could only be achieved in the range of low communications as stated above.

We conclude that the processor-driven technique is not adequate to implement global priority-based task selection in the general case because it does not provide fair balancing between task-criticalty and local processor idle time minimization.

4.5.3 Local Heuristics

In general, the local heuristics PD/ETF and GD/ETF have nearly the same average deviation from ω_{best} in all cases except in the range of medium to high communication ($1 \leq \alpha \leq 3$) where GD/ETF outperforms PD/ETF by 2%. The reason for the improvement is the use of the effective earliest-starting-time (Eq. 6) by GD/ETF that is more accurate than that used by PD/ETF (Eq. 5). The effective $est(T)$ is more appropriate as the basis for task-selection because it gives higher priority to the task that can effectively start at the earliest, i.e. to minimize the effective processor idle time.

Increasing the parallelism in the computation leads to improved performance because local heuristics have more opportunity to overlap computation with communication when parallelism increases. This effect is shown in figure 4.6 for the FC topology where the least average deviation (about 5%) is obtained for the

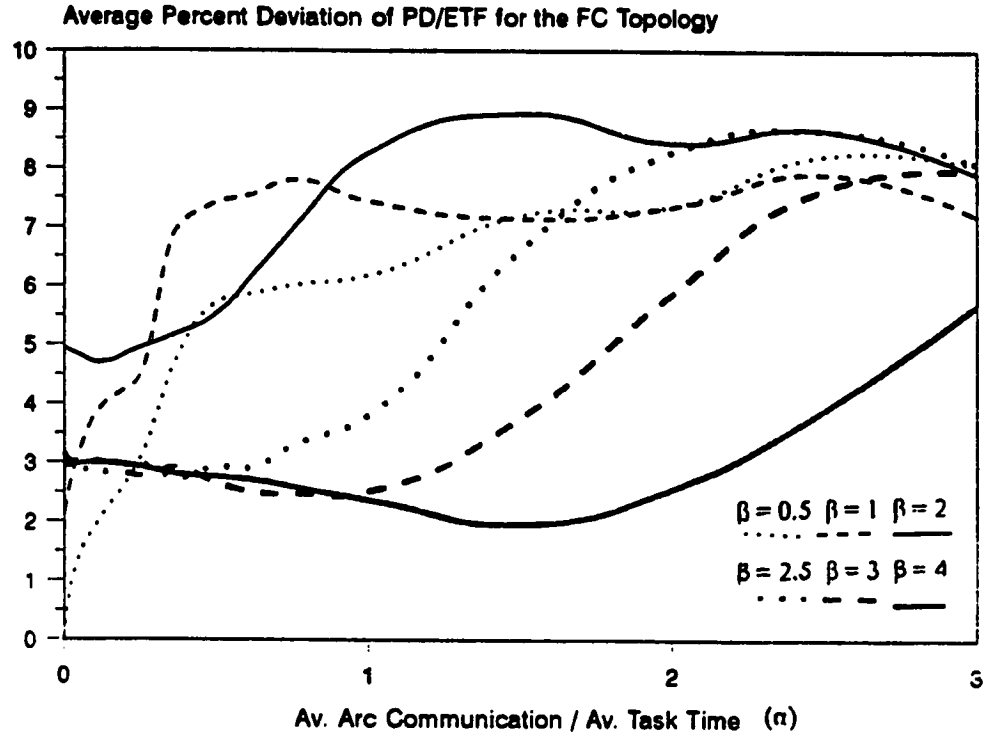

Figure 4.6: PD/ETF performance is acceptable only when $\beta/\alpha \geq 1.5$ for FC.

highest amount of inherent parallelism ($\beta = 4$). However, for low to medium graph-parallelism ($0.5 \leq \beta \leq 3$), these heuristics significantly deviate from ω_{best} by 9% (FC), 10% (HC), 12% (RG). In this range of graph-parallelism, there are generally few ready-to-run tasks that compete for every processor. Therefore, the opportunity of locally maximizing the processor efficiency, that is the main strategy of PD/ETF and GD/ETF, is reduced. These local heuristics have reasonable deviation only for high graph-parallelism provided that the communication requirements and the interprocessor communication penalty are moderate. Increasing graph-communication with the HC and RG topologies removes any potential improvement even for high degree of graph-parallelism. Figures 4.7 and 4.8 show that in the range of communication $1.5 \leq \alpha \leq 3$, the deviation of PD/ETF and GD/ETF is about 9% even with high graph-parallelism ($\beta = 4$).

Using the plots on figures 4.6, 4.7, and 4.8, acceptable finish time can be achieved by the local heuristics PD/ETF and GD/ETF only when the amount of available parallelism is sufficient to cover the average communication. In other words, local heuristics nearly deviate by 5% from ω_{best} when the ratio β/α is 1.5 (FC), 2.3 (HC), and 4 (RG), i.e. β/α should be greater than a topology-dependent factor ϵ_{top}: $\beta/\alpha \geq \epsilon_{top}$. Using the definition of α and β from the previous section, we have:

$$\frac{N_T}{N_L} \cdot \frac{\mu_T}{C_{arc}} \geq \epsilon_{top} \cdot p$$

To achieve acceptable performance under a given amount of inherent parallelism (N_T/N_L) and communication (C_{arc}/μ_T), the local heuristics (PD/ETF), (GD/ETF) impose a bound ($\epsilon_{top}p$) on the number of processors used, and as a result, bounded speedup is the only alternative to maintain acceptable performance for these local heuristics.

We conclude that local heuristics (PD/ETF),(GD/ETF) that attempt minimizing the processor idle times by using the notion of earliest-task-first give ac-

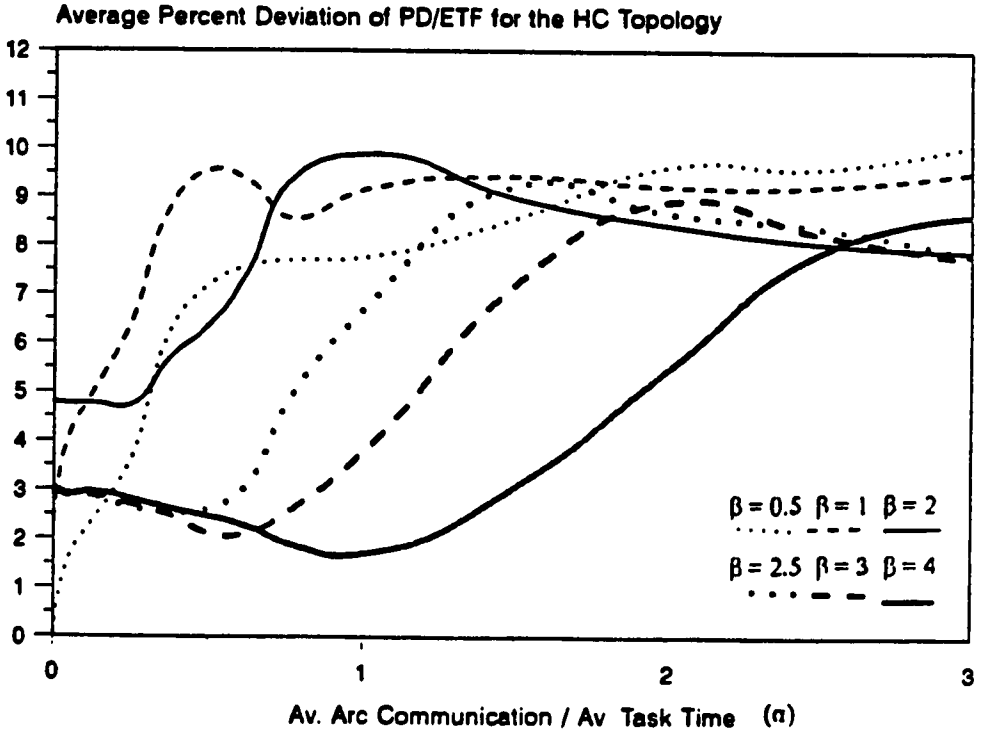

Figure 4.7: PD/ETF performance is acceptable only when $\beta/\alpha \geq 2.3$ for HC.

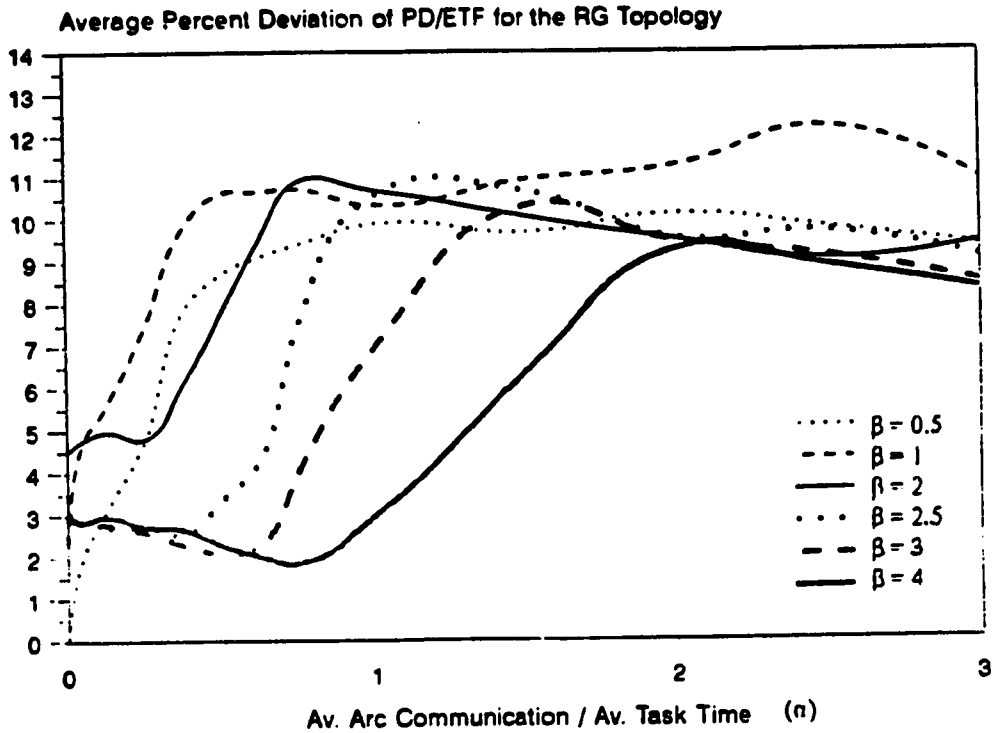

Figure 4.8: PD/ETF performance is acceptable only when $\beta/\alpha \geq 4$ for RG.

ceptable performance when: 1) the amount of parallelism in the computation is much higher than that of the available processors, i.e. for high values of β ($\beta = 3.5, \beta = 4$), and 2) the inherent communication requirements and the inter-processor communication penalty are moderate.

4.5.4 Generalized List Scheduling

In this section we analyze the performance of graph-driven HLF, $HLETF$, HLF^*, and $HLETF^*$ which all use the global priority $lst(T)$. All these heuristics perform well in the range of low to medium communication ($0 \leq \alpha \leq 1.5$). Figures 4.9 to 4.15 show the average finish time achieved by heuristics $GD/HLETF$ and $GD/HLETF^*$. The Performance of GD/HLF and GD/HLF^* are qualitatively similar to those of figures 4.9-4.14 but differ in the relative deviation which we discuss next.

Increasing the communication or/and increasing the interprocessor communication penalty (FC, HC, RG) lead only a to slight degradation compared to that of the local heuristics. GD/HLF is performing last with respect to GLS/Graph-driven group especially in the range of high communication. This suggests that task-selection according to $lst(T)$ alone suffers from deficiency in managing the processor idle times. There are two reasons for this effect: 1) the principle "highest $lst(T)$ first" lacks control of the processor efficiency, and 2) heuristic measurement of task-priority $lst(T)$ by using algorithm lst slightly loses accuracy with increasing graph and interprocessor communication.

Heuristic $GD/HLETF$ applies the principle highest "$lst(T) - est(T)$" first and gives sharp improvements (8% for FC, 5% for HC, and 9% for RG) over GD/HLF specially in the range of high communication. On the other hand, efficient management of idle time can also be achieved through the use of the optimization technique, i.e. use of GD/HLF^*. Heuristic GD/HLF^* significantly

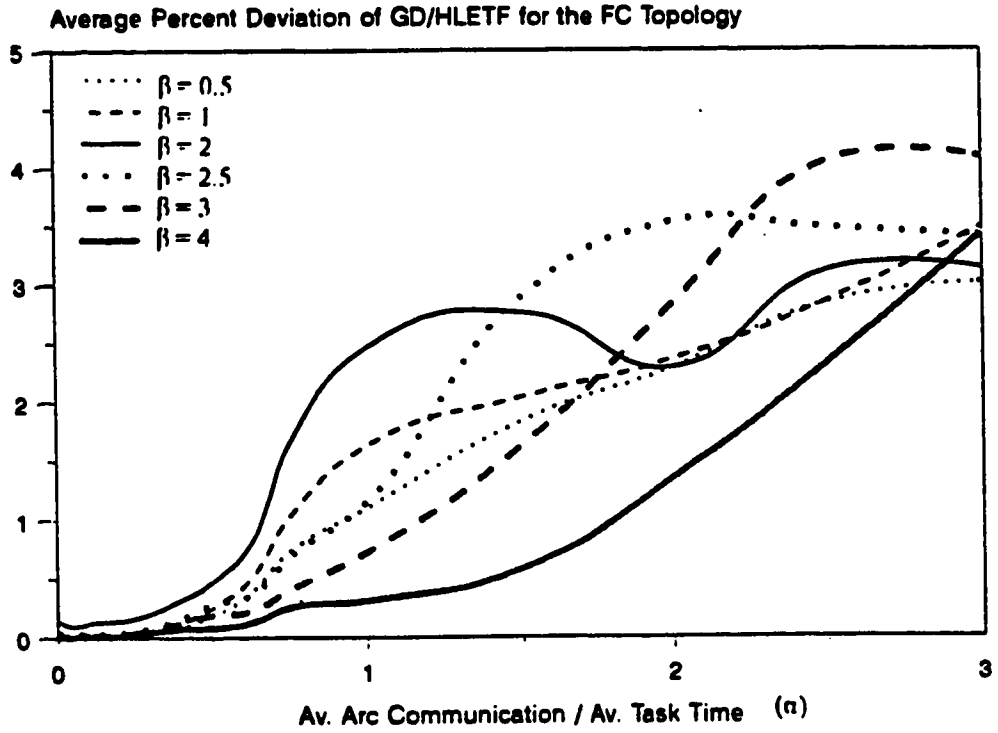

Figure 4.9: The relative performance of $GD/HLETF$ for the FC topology.

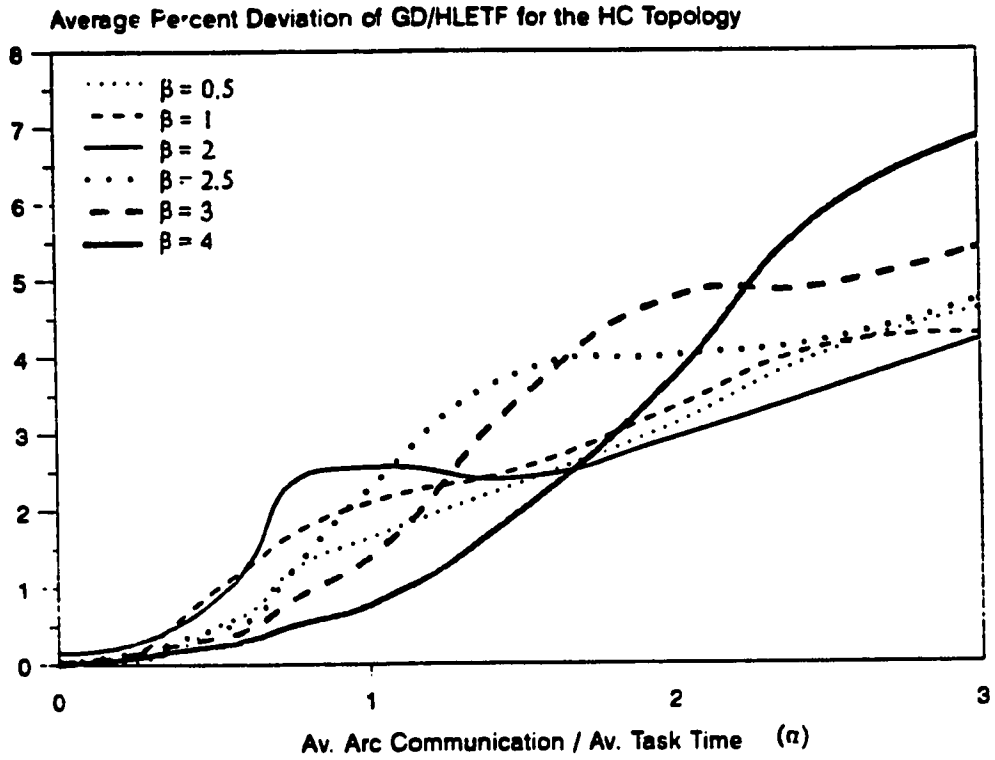

Figure 4.10: The relative performance of $GD/HLETF$ for the HC topology.

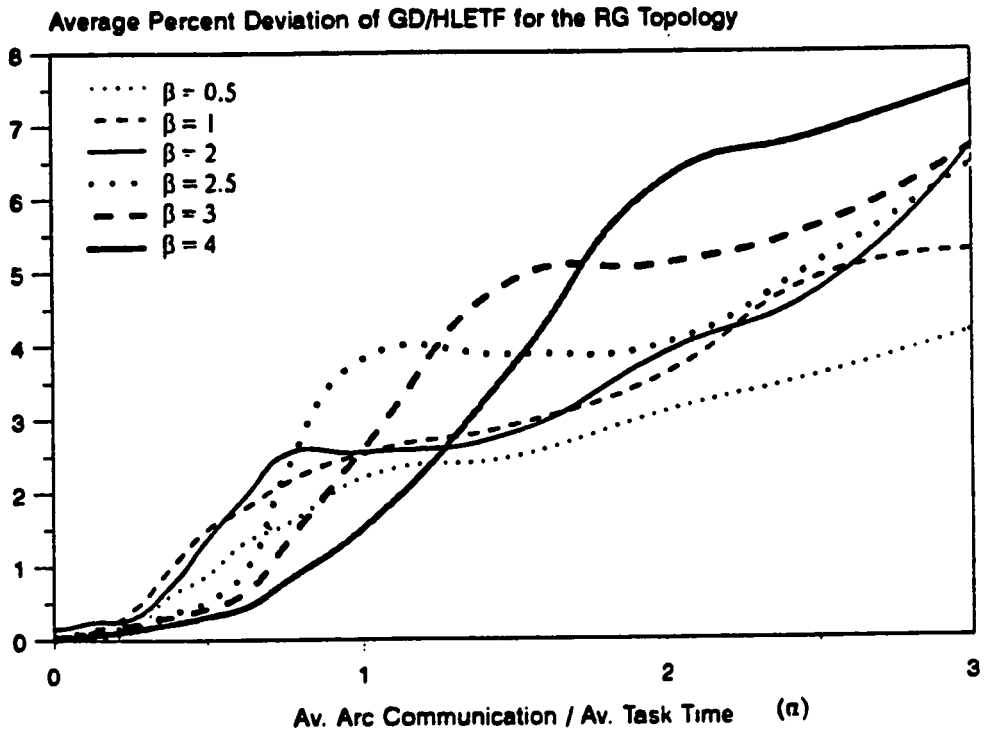

Figure 4.11: The relative performance of $GD/HLETF$ for the RG topology.

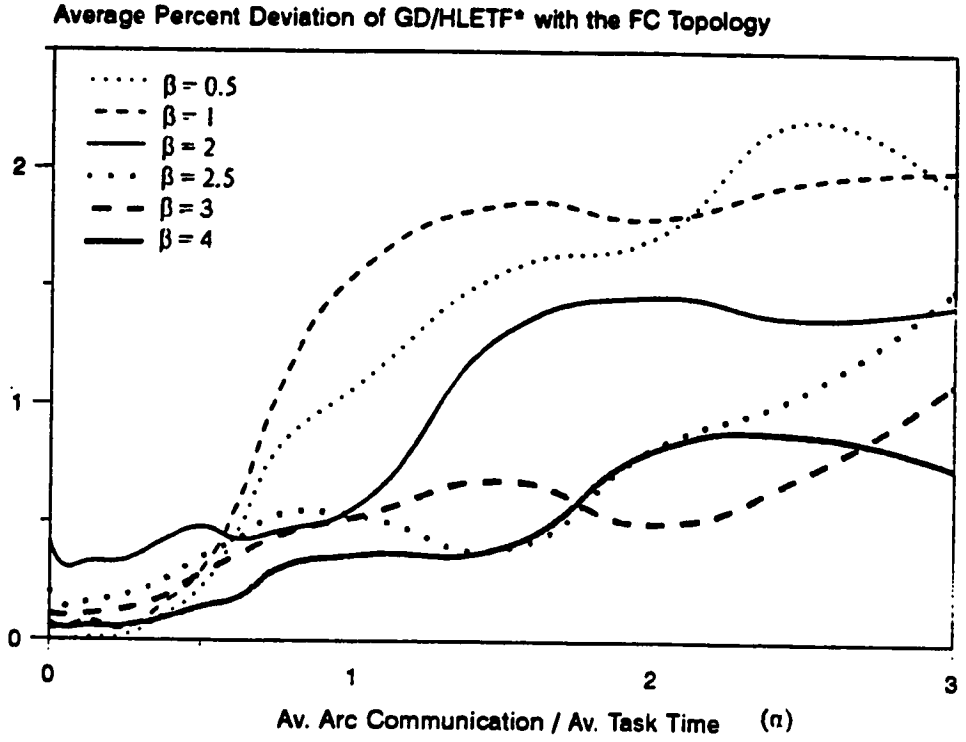

Figure 4.12: The relative performance of $GD/HLETF^*$ for the FC topology.

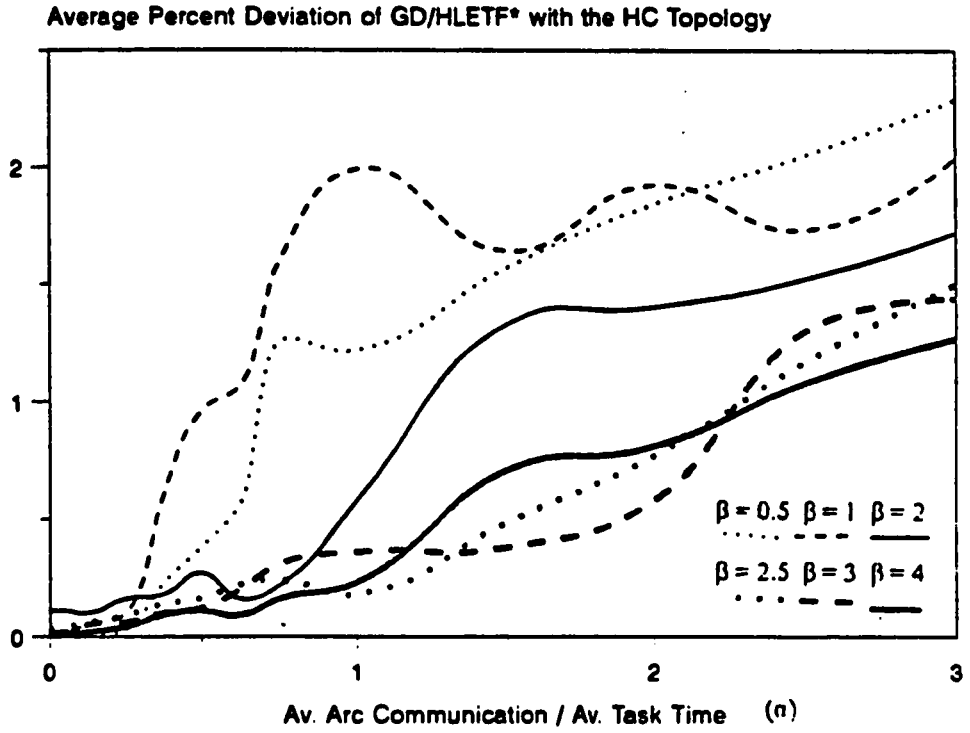

Figure 4.13: The relative performance of $GD/HLETF^*$ for the HC topology.

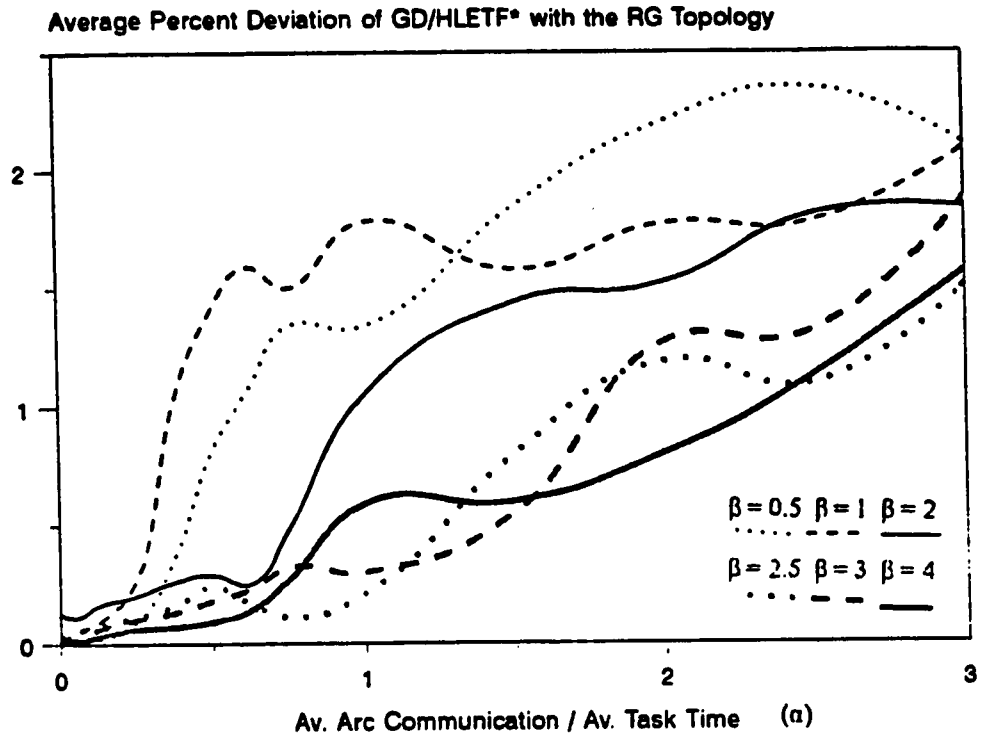

Figure 4.14: The relative performance of $GD/HLETF^*$ for the RG topology.

improves the performance of GD/HLF by 6% for FC, 10% for HC, and 13% for RG. Therefore, removing most of the deficiency of GD/HLF, with respect to the management of the idle times, can then be made by the use of either $GD/HLETF$ or GD/HLF^* with a slight advantage (1%) to GD/HLF^*.

Finally, by combining both optimization technique and the $est(T)$ penalty, as this is done with heuristic $GD/HLETF^*$, an improvement of 7% for FC, 11% for HC, and 13% for RG over GD/HLF has been achieved. Heuristic $GD/HLETF^*$ gives the best average deviation among all the studied global and local heuristics. It outperforms $GD/HLETF$ and GD/HLF^* by 2% and 1% in all cases, respectively. Sorting the GLS heuristics in the order of best-to-worst is: $GD/HLETF^*$, GD/HLF^*, $GD/HLETF$, and GD/HLF.

We also note that these heuristics nearly maintain their relative performance when switching from the FC to HC and to RG topologies. The reason is that algorithm *lst* evaluates the priority $lst(T)$ by using the task-graph and the specific multiprocessor system. Therefore, the task-level is topology dependent. Analysis of the GLS group suggests that $lst(T)$ was critical for achieving the least average deviation. This information allows maintaining coherent performance under different inherent parallelism and topologies.

Figure 4.15 shows the results obtained with the PD/HLF heuristic. The heuristic performed the worst among all the other tested ones, even worse than random. As mentioned before, Random is a graph-driven heuristic that introduces randomness only in the tasks selection process. Otherwise, tasks are always scheduled at earliest and successors expansion is exactly the same. The fact that PD/HLF performs even worse than $GD/Random$ is because PD/HLF schedules tasks with high *lst* values first, thus sacrificing local optimization of processors free times done by PD/ETF. However, because it is event driven, it does not allow the successors of these critical tasks to be expanded and scheduled. As a result, the purpose of using *lst* is wasted. Moreover, since local time management is not

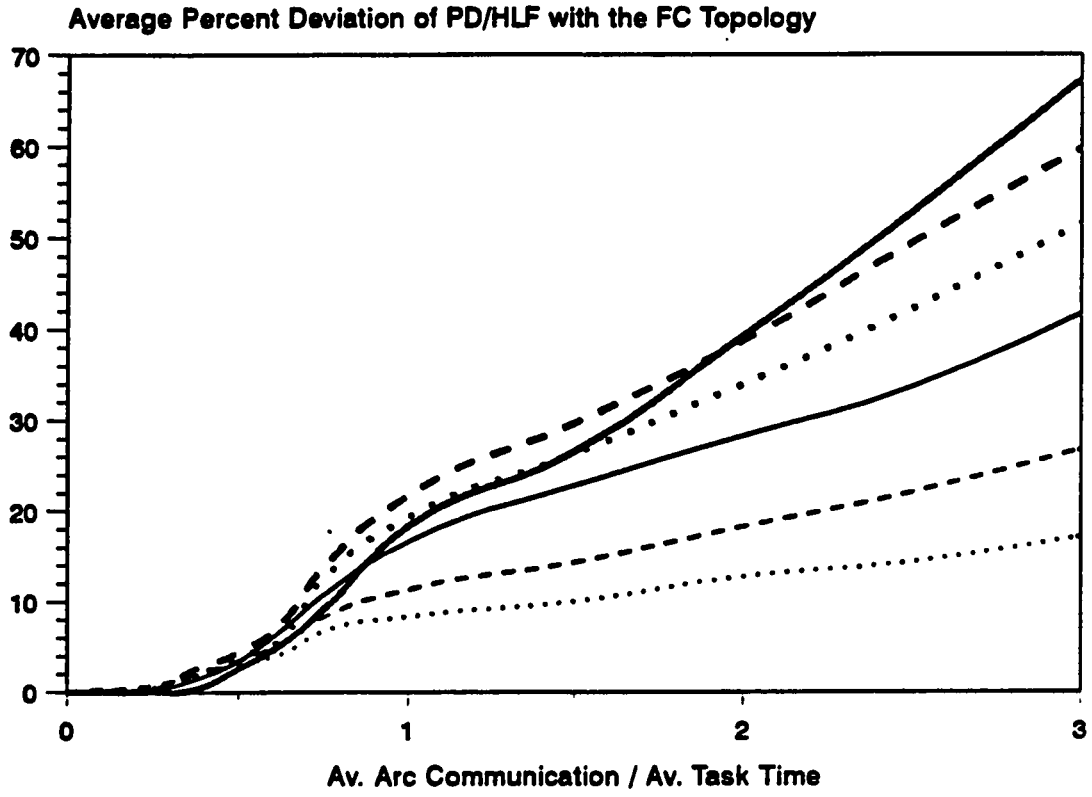

Figure 4.15: The relative performance of PD/HLF for the FC topology.

used in PD/HLF, no hole filling of idle times is done. The effect is a local heuristic modified to use a global measure. The resulting heuristic does not exploit the main features of both worlds. Although, GD/Random selects tasks at random, the fact that it allows vertical expansion gives a better chance for critical successor tasks to be scheduled early.

The graph-driven approach and the notion of global task-priority appear to be more efficient than the processor-driven and the earliest-task-first criteria for achieving the objective function under the tested conditions. The graph-driven strategy leads to early scheduling of critical tasks along their paths compared to local heuristics. Implicitly, they expand the task-graph depth-first until a blocking occurs due to precedence constraints. Less important tasks are then scheduled until the blocking is removed and so on. This technique works well even with estimated $lst(T)$ times. Another interesting aspect is the ability of either $GD/HLETF$, GD/HLF^*, and $GD/HLETF^*$ in beating PD/ETF and GD/ETF even where these local heuristics give their best performance for high degree of parallelism. This seems to be very useful in minimizing the processor idle times, because even for high values of β, the GLS heuristics outperform the local heuristics whose strategy is based on local maximization of the processor efficiency.

4.5.5 Analysis of the distribution

For each instance of α and β and for each topology, 500 graphs are randomly generated and the finish time obtained from each heuristic is recorded. The plot of the most relevant average deviations are shown on Figures 4.4 to 4.14. Typically, the average finish times of the heuristics represent the boundary of the best 50% to 65% for all the studied heuristics except for *Random* which exhibited a flatter distribution.

Analysis of the distribution is carried out by considering the PD/ETF and

$GD/HLETF$ heuristics because these heuristics are representative of local and GLS scheduling, respectively. Figures 4.16 and 4.17 show the boundary of the best 50% and 90% finish times for PD/ETF and $GD/HLETF$, respectively. These boundaries represent the maximum deviation among all levels of communications for a given amount of parallelism (β) and a given topology. Therefore, the boundaries represent experimental worst case performance for each studied instance of parallelism and topology.

The 50% boundary on Figure 4.17 shows that $GD/HLETF^*$ has a distribution that is strongly concentrated within 0.5% deviation from ω_{best}. Heuristic PD/ETF has much flatter distribution than that of $GD/HLETF^*$ as its 50% boundary (figure 4.16) is nearly at the 9% level.

The best 90% of the graph-runs may deviate up to 7% and 20% for PD/ETF and $GD/HLETF^*$, respectively. However, the percent deviation between the 90% and 50% boundaries are nearly the same for both heuristics.

Heuristic PD/ETF is more sensitive to the amount of inherent parallelism than $GD/HLETF^*$. While PD/ETF improves its average finish time versus increasing parallelism ($0.5 \leq \beta \leq 4$), $GD/HLETF^*$ maintain constant performance at the 50% boundary for different parallelism and topologies. The dependency on parallelism and topology appears only at the 90% boundary for $GD/HLETF^*$.

For other multiprocessors, the effect on the distribution is that restricted connectivity (HC, RG) with respect to FC leads mainly to increasing the communication requirements for the studied heuristics and requires sophisticated task-to-processor assignment because of increasing communication penalty. Testing with the Hypercube and Ring naturally leads to increasing the variance on all the heuristics but, qualitatively, the above comparison between the distribution of the heuristics remains valid as shown for $GD/HLETF$ and PD/ETF in figures 4.16 and 4.17.

Testing the effect of the task graph variance on the heuristic scheduling has also

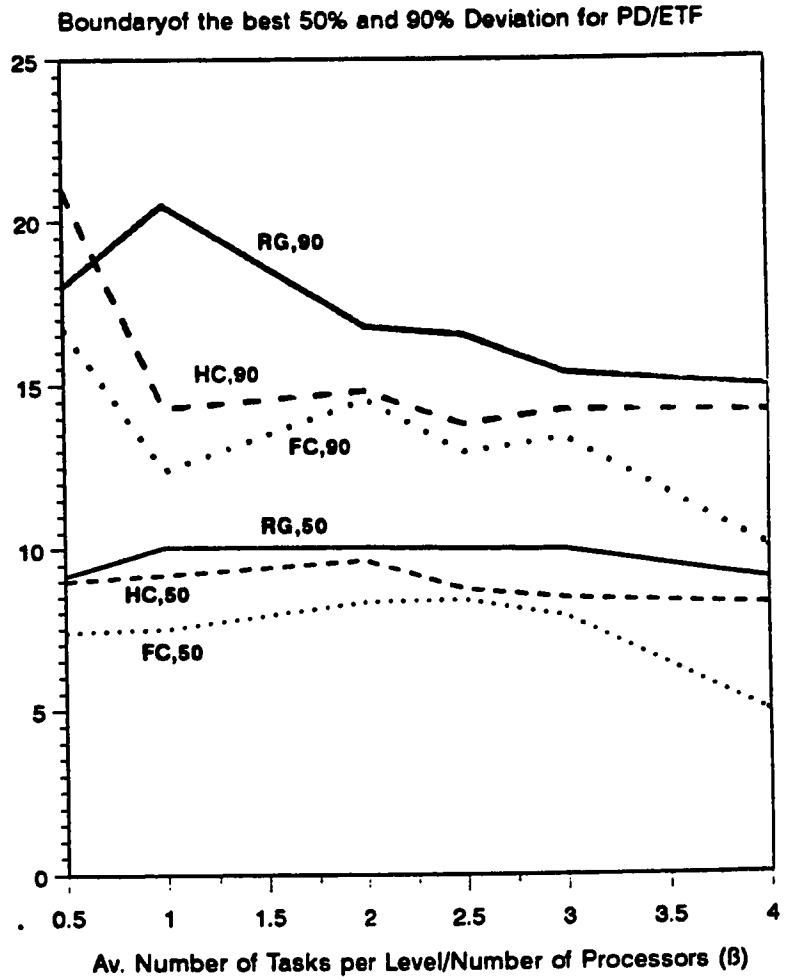

Figure 4.16: Maximum deviation of PD/ETF from ω_{best} under each instance of α, β and topology.

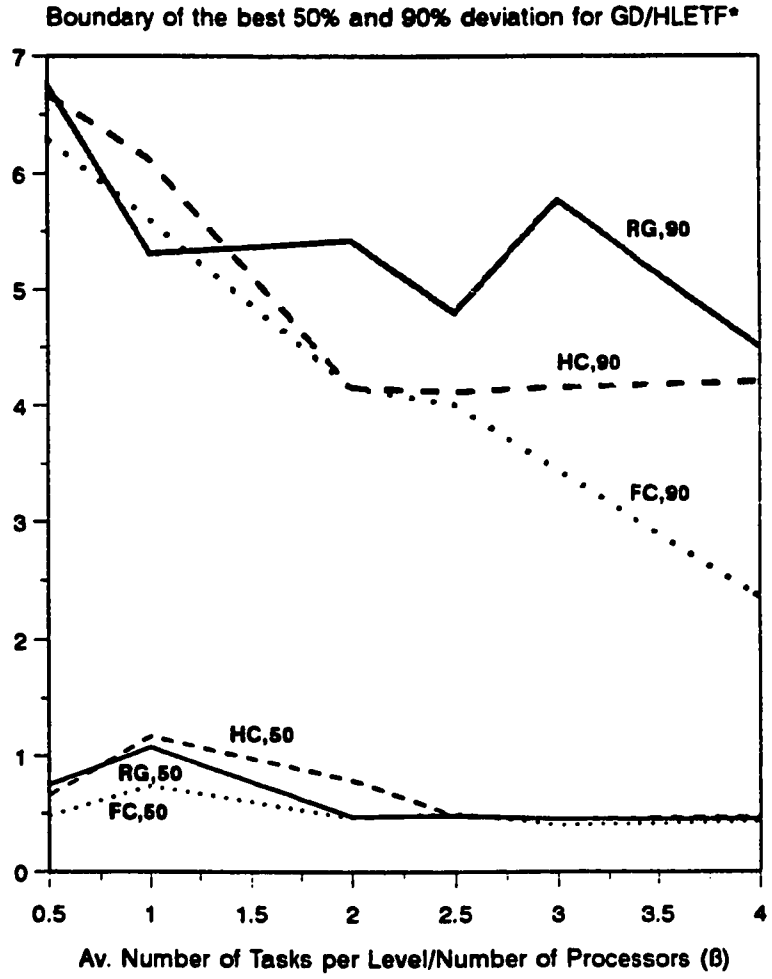

Figure 4.17: Maximum deviation of $GD/HLETF^*$ from ω_{best} under each instance of α, β and topology.

been carried out by decreasing the variance of the task time and communication with respect to their values as defined in the previous section. Using two different experiments, the variances of task time and communication are dropped to 50% and 10% of their original values and heuristic scheduling is performed as before. Analysis of the scheduling performance shows that the results outlined in this section are valid within at most 1% deviation under the reduced variances. This reinforces the performance analysis outlined in this section.

Chapter 5

Optimizing GLS Heuristics

In this chapter, we present a general optimization technique for optimizing global list scheduling heuristics. The technique is a low cost iterative method that imposes a linear effect on the complexity of the heuristic it is applied on. The results obtained from applying this technique to some global heuristics studied in the previous chapter are also presented.

5.1 Task Level Revisited

In the previous chapter, we defined the latest starting time of a task $lst(T)$ as the priority criterion used for task selection. The task with the highest lst value is the task with have highest priority. The lst value is sometimes referred to as the task-level $l(T)$[1], since it is actually an estimate of the shortest possible path from starting T to some terminal task $T\prime$. The task-level or lst as defined previously accounts for all the computation and some communication arcs along the directed path $\gamma = (T \rightarrow \ldots \rightarrow T\prime)$.

[1]The terms $lst(T)$ and $l(T)$ are used interchangeably

For the model with no communication $G(\Gamma, \rightarrow, \mu)$, the communication along the path γ is ignored. Thus $lst(T)$ is evaluated as follows:

$$lst(T,p) = \mu(T) + \begin{cases} 0 & if succ(T) = \phi \\ \max\{lst(T_i) : T_i \in succ(T)\} & \text{otherwise} \end{cases}$$

Applying this definition to the general $G(\Gamma, \rightarrow, \mu, c)$ model leads to unrealistic estimation of the task-level and as a result the task selection criteria which is based on highest $lst(T)$-first becomes too inaccurate. On the other hand, if we account for all the communications along γ in evaluating lst, we would have,

$$lst(T,p) = \mu(T) + \begin{cases} 0 & if succ(T) = \phi \\ \max\{lst(T_i) + c(T,T_i) : T_i \in succ(T)\} & \text{otherwise} \end{cases}$$

The above formula also leads to an inaccurate estimation of the task-level. This is due to the fact that the above estimation equation does not reflect the task assignment to processors, i.e. the factor $r(p(T), p(T_i))$ is not accounted for.

For certain processor topologies, the processor to processor communication cost varies widely. Therefore, accounting for the inter-tasks messages alone independently of the processors topology does not present a true realistic estimate of the task level. The resulting schedule would fail to identify true critical tasks in most cases.

In the previous chapter we presented an approach to estimate a realistic task-level value. The approach is based on scheduling the reverse graph first using a local heuristic. The resulting completion times of the reverse-graph tasks are used as task-level values when scheduling the original graph. To better understand this process, we need to understand the completion time of a task in the reverse graph and how it can be used as a task level or priority measure in the actual graph.

5.2 Completion time and Task-level

As explained before, the task-level is an estimate of the distance from the task to the end of the schedule. The task-level values are used to distinguish critical from less critical tasks by giving more priority to those tasks with higher *lst* values. When scheduling the reverse graph using a local heuristic, the scheduler tries to minimize the idle times of processors by scheduling ready-to-run tasks with earlier starting times before other ready tasks with later starting times. As a result, the process can be looked upon as an attempt to "Squeeze" the graph along the time axis in the direction of time 0 (towards the start of the schedule). Several factors determine how early a task can start. These factors are :

- The precedence constraints that delay the start of a task until all its predecessors have finished their execution.
- Messages sent from predecessors which delay the start of a task until it has received all of them from all sending predecessors. Local heuristics try to schedule tasks at earliest by assigning the selected task on the processor that enables the task to receive all it's message as early as possible and start at the earliest.
- Processors idle times : meaning that even if a task has satisfied all its precedence constraints and have received all message from its successors, it can only start, at least, when one of the processors becomes idle.

we shall refer to these factors as *delay* factors. When scheduling the reverse graph using a local heuristic, the *delay* factors determine how early tasks can start. Tasks with less *delay* effect can start earlier than tasks with higher *delay* effect. The resulting tasks completion times of this reverse schedule become a measure of the distance between the start of the schedule and the end of the tasks.

When scheduling the original graph using tasks global priorities as in previous chapter, we used the completion times of the reverse graph as the *lst* values. Regarding the original graph, these values now represent the distance between the starting time of the tasks to the end of the schedule. The longer this distance is, the more important the task becomes. This is because the *delay* factors that had affected the starting time of the task in the reverse graph become the *delay* factors that would affect the starting time of the successor tasks in the original graph. Also, the finish time of a task T in the reverse graph represents an achievable realistic measure of the shortest possible distance from T till the start of the schedule. This distance accounts for all the tasks along the chain of grand fathers and fathers of T, the messages transferred to T and the processors assignments. As a result, it gives an indication, in the actual graph, of what lies ahead after scheduling T and how much is left, along the path that T resides on, until the end of the graph. Consequently, lower *lst* values indicate less *delay* effects ahead and thus delaying their schedule would result in less global effect on the finish time than delaying tasks with higher *lst* values.

Several tests were conducted to strengthen this point. Referring to the scheduling tests on various graphs and topologies explained in previous chapter, some samples of the same tests were conducted using the *lst* values obtained from the following methods:

- Scheduling the reverse graph on infinite number of processors with identical links and using the completion times of tasks as *lst* values (neglecting the effect of processors topology).

- Scheduling the reverse graph on infinite number of processors with zero- cost links and using the completion times of tasks as *lst* values (neglecting the effect of processors topology and the inter-tasks communications).

Comparing the resulting schedules length with the original ones (which used the completion time in the reverse graph as *lst* values) show that these two methods resulted in longer finish times (2% to 7%) than the one proposed. Therefore incorporating the graph and topology effects in calculating the *lst* values gives a better estimate of the task-level and thus leads to improved management of tasks priorities.

5.3 Forward-Backward Scheduling

An approach to evaluate an accurate estimate of task-level has been described in the previous chapter, which relies on the pre-scheduling of the reverse task-graph $G(\Gamma, \rightarrow, \mu, c)$ by using the best known local heuristic. In the rest of this chapter, the computation graph $G(\Gamma, \rightarrow, \mu, c)$ and its reverse graph $G_R(\Gamma, \rightarrow, \mu, c)$ will be denoted by G and G_R, respectively. The best local heuristic is earliest-task-first that attempts to minimize the global finish time by locally minimizing the processor idle times. Let H_{loc} be the local scheduling heuristic that schedules computation G_R on target message-passing system $S(P, R)$ and yields the mapping of every task T on some processor $p(T)$ together with its completion time $ct(T)$:

$$H_{loc}(G_R, S(P, R)) = \{(ct(T), p(T)) : T \in \Gamma\}$$

The task completion time $ct(T)$ is considered as an approximation of the task-level $l(T)$ because heuristic H_{loc} attempts to minimize the time distance from the start of any task to some terminal task with respect to the reverse graph. This amounts leads to incorporate the effect of the computation, communication, and the topology of the multiprocessor in evaluating the task- levels. The use of $l(T)$ within global priority-based scheduling heuristics is useful to distinguish critical from non-critical tasks, therefore, leading to a better schedule.

Let H_{glb} be a global priority-based scheduling heuristic for which the task

selection criteria is highest $\alpha(T)$ value first, where $\alpha(T)$ is some increasing function of the task-level $l(T)$. The function $\alpha(l(T))$ may be designed by combining the task-level with other parameters. For example, the heuristic highest-level-earliest-task-first ($HLETF$) selects the task whose $\alpha(T) = l(T) - est(T)$ is highest among all the ready-to-run tasks, where $est(T)$ is the effective earliest- starting-time of T for some idle processor p. Denote by T' any currently ready-to-run task other than T. Heuristic $HLETF$ selects task T first if the difference in levels $l(T) - l(T')$ is greater than the amount of effective processor idle time $est(T) - est(T')$ that would be saved if T' was selected first. The global priority-based scheduling heuristic $HLETF$ combines both task-criticality concept with management of the processor idle times in order to achieve the objective function. Experimental results indicate that global priority based scheduling is always superior to the best local scheduling under different instance of communication levels, inherent parallelism, and multiprocessor topologies.

The scheduling of a computation graph G onto a message-passing system $S(P, R)$ consists of the following steps:

1. Evaluate the reverse graph G_R.

2. Apply heuristic scheduling of G_R over system $S(P, R)$ using a local scheduling heuristic $H_{loc}\{G_R, S(P, R)\} = \{(ct(T), p(T)) : t \in \Gamma\}$ and obtain the list of task-levels $L = \{ct(T)\}$.

3. Schedule G onto multiprocessor $S(P, R)$ using the global priority- based heuristic $H_{glb}(G, S(P, R), L) = \{(ct(T), p(T)) : T \in \Gamma\}$ whose performance function is the finish time $\omega = \max\{ct(T) : T \in \Gamma\}$.

This scheduling procedure appears to be a one-step refinement process on the global finish time because the achieved task finish times, which result from scheduling the reverse graph, are passed to the global priority-based forward scheduler as

task-levels. The later scheduler improves the global finish time because the available task-levels represent combined information on task criticality with respect to the computation graph and the system $S(P,R)$. Clearly, this scheduling approach can be seen as based on one-step backward (Scheduling of G_R) and forward (Scheduling of G) over the system $S(P,R)$. A natural extension to this approach would be to iterate this process several times so as to improve the overall accuracy of task priorities and by the same time the length of the schedule. This process of iterative backward/forward scheduling, in which passing the task-levels from one iteration to the next is the key to improving task-levels estimation, consequently to obtains schedules with shorter global finish time.

1. Initialize the priority list by backward scheduling:

   1.1 Evaluate the reverse graph G_R by reversing all the arc directions in G.

   1.2 Schedule G_R on system $S(P,R)$ by using the local heuristic $H_{loc}(G_R, S(P,R)) = \{(ct(T), p(T))\}$

   1.3 Obtain the priority list: $L = \{ct(T)\}$ and initialize counter $i = 0$.

   Repeat

2. Perform forward scheduling using the global priority-based heuristic H_{glb} : $H_{glb}(G, S(P,R), L) = \{(ct(T), p(T))\}$,

   Update the priority list: $L \leftarrow \{ct(T)\}$,

   Evaluate the performance function: $\omega_F = \max_{T\in\Gamma}\{ct(T)\}$.

3. Perform backward scheduling using the global priority-based heuristic H_{glb} : $H_{glb}(G_R, S(P,R), L) = \{(ct(T), p(T))\}$,

   Update the priority list: $L \leftarrow \{ct(T)\}$

   Evaluate the performance function: $\omega_B = \max_{T\in\Gamma}\{ct(T)\}$,

Find best finish time at current iteration: $\omega(i) = \min\{\omega_F, \omega_B\}$.

4. Update iteration counter: $i = i + 1$ Until condition $f(|\omega(i-k) - \omega(i)|) \leq \epsilon$ is satisfied, where k is some integer and ϵ is some positive number.

A task graph has a set of entry tasks $\{T_e\}$ and a set of terminal tasks $\{T_t\}$ that are connected through arbitrary paths. For the forward task-graph, each terminal task T_t is connected to a sub-set of entry tasks through different paths. However, the finish time of each terminal task critically depends on one or more entry tasks. For a given multiprocessor topology, the schedule that has the least finish time ω_{opt} (among all the other iterations) is characterized by one or more critical paths whose terminal tasks complete execution at time ω_{opt}, as well as a number of secondary paths. Delaying a critical task over its earliest starting time leads to an increase in the overall schedule length. Secondary tasks, however, can be delayed by some time without causing any delay (or a comparably slight delay) on the overall finish time. Global priority-based scheduling leads to select the tasks according to the greedy policy "highest-level-first" (or highest priority first). Therefore, increasing the level of a task means that the task is given more opportunity to start at the earliest time on the most suited processor.

5.4 Analysis of the Forward-Backward Mechanism

The forward/backward strategy is based on increasing the task-level in the next scheduling pass for those tasks whose delay in the current pass causes an increase to the finish time of the schedule. Consequently, the priorities of less critical tasks will be decreased.

Each iteration in the Forward-Backward scheduling involves scheduling the

G_R graph followed by scheduling the G graph. The objective is to find, at one of these iterations, schedules with lower finish time. This process does not cause any change to the graph G or the processors topology. The only varying scheduling factor that changes in each iteration is the tasks *lst* values. Each iteration delivers its tasks completion times to the next iteration as tasks *lst* values. To understand the internal mechanism of this process and how it investigates possible better scheduling solutions, we consider the task graph of figure 5.1 . The figure shows a schematic of the graph with T_a and T_b as internal tasks (i.e both tasks are not entry or exit nodes). We shall assume that the precedence structure of the graph permits the two tasks to coexist at the same time in the set of ready to run tasks when scheduling G [2].

Each of $chain_a^1$ and $chain_b^1$ represent the most critical chain of predecessors for the tasks T_a and T_b respectively [3]. Also, Each of $chain_a^2$ and $chain_b^2$ represent the most critical chain of successors for the tasks T_a and T_b respectively. We will assume that $chain_a^1$ and $chain_b^1$ both have the same length. We shall also assume that $chain_a^2$ is substantially shorter than $chain_b^2$ meaning that T_b resides on a more critical path than T_a's. As a result, a good schedule should give higher priorities for T_b and $chain_b^2$ compared with T_a and $chain_a^2$. As a final assumption, we will start the forward-backward scheduling with T_a and $chain_a^2$ having higher priorities than T_b and $chain_b^2$. This assumption is meant to introduce inaccuracy into task-priority and the initial schedule so as to prolong it. The iterative forward-backward process should then be able to detect this inaccurate priority assignment and adjust the *lst* values to reflect a more realistic task level values. The following describes the scheduling process through the forward-backward iterations:

[2]The assumption is valid because we can always arrange the graph G so that T_a and T_b would fall into the same set of ready to run tasks regardless of their priorities.

[3]A critical chain of predecessors for a task T is actually the part of the critical path, on which T resides, from the entry node to T.

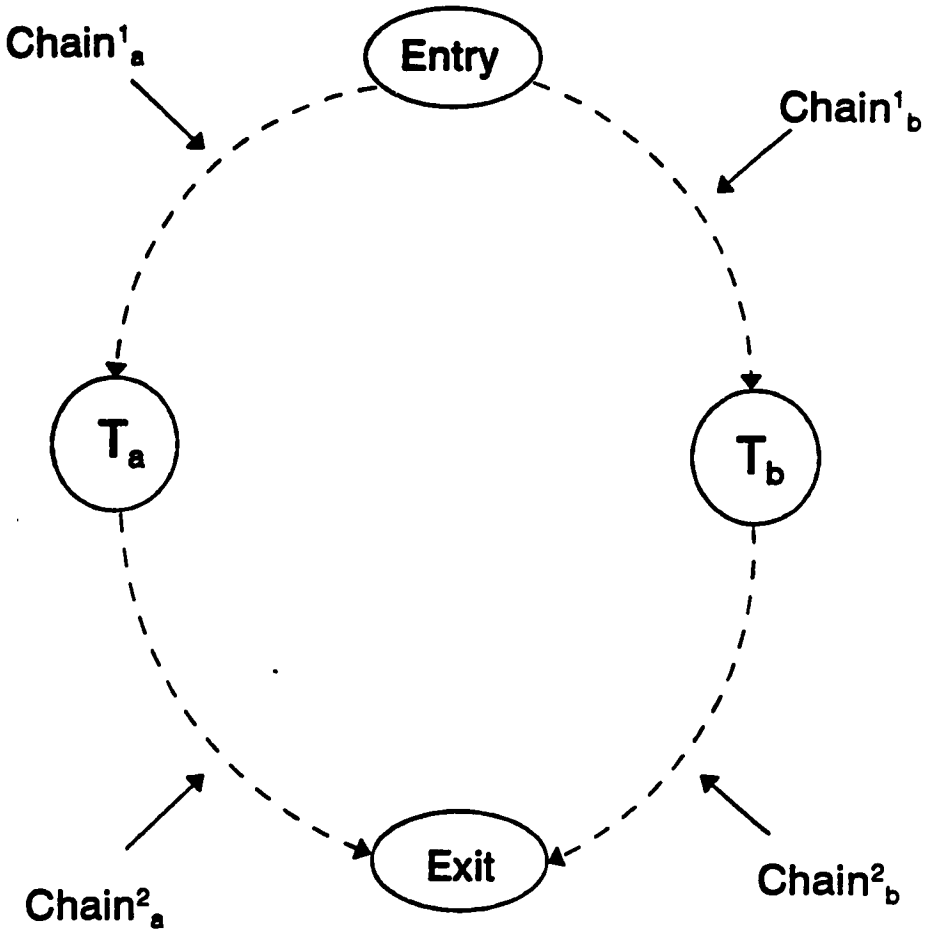

Figure 5.1: Internal chains of tasks in a task graph

- *Backward scheduling* of G_R in order to establish the initial *lst* values for the original graph G. The *lst* values of the task T_a and $chain_a^2$ are increased to give them higher false priority.

- *Forward scheduling* of G. The tasks T_a and T_b will fall into the same set of ready to run tasks and thus will compete for the resources. Since T_a has a substantially higher priority than T_b, it will be scheduled before. The same applies for the $chain_a^2$ and $chain_b^2$ tasks. With the assumption that T_a and T_b have close execution times, T_a will finish earlier than T_b and thus will have less *lst* value in the next backward iteration.

- *Backward scheduling* of G_R. Since $chain_a^2$ is shorter than $chain_b^2$, T_a will become ready for scheduling earlier than T_b. Thus, even though T_b has higher *lst* value than T_a, it will probably be scheduled after T_a since all the tasks of the longer $chain_b^2$ will be scheduled before. It's worth mentioning that although the tasks along $chain_b^2$ have higher *lst* values than T_a and the tasks along $chain_a^2$, the fact that these later tasks will be ready for scheduling earlier than their opposite tasks in the other chain [4] permits them to hole fill idle times that the tasks of $chain_b^2$ can't fill because of their communications and precedence constraints. As a result, several tasks along $chain_a^2$ and T_a itself will be scheduled early enough. Thus, they will have comparably low completion times and lower *lst* values for the next forward schedule than the one they had in the backward schedule of *iteration 1* (where they were assigned false high *lst* values).

- *Forward scheduling* of G. Since T_a and the tasks of $chain_a^2$ have less *lst* values than the ones in the forward schedule round of *iteration 1*, the more important T_b task and the $chain_b^2$ tasks will have more priority in scheduling

[4] Those tasks that would most probably fall into the same set of ready to run tasks

and thus less chance to be delayed and delay the whole schedule.

Therefore, the correction on the task-level evaluation continues from one iteration to another until one of the following occurs:

- The schedule finish time converges to some value after a number of refinements. This means that regardless of how many more iterations are applied to the schedule, the finish time stays the same. Through experimental testing, it was found that the value the schedule finish time converges to may or may not be the minimum among all schedules. Figure 5.2 shows the case when the iterative improvement technique is applied to the $GD/HLETF^*$ heuristic for the fully connected architecture. For this case, the technique converges to the lowest value of all iterations. Figure 5.3 shows the case where the mechanism converges to some value which is not the minimum.
- The process oscillates and does not converge (see figure 5.4).

As seen from the figures, the number of iterations was limited to 100 iterations only. Figure 5.5 shows the case where 100 iterations where not enough to lead to convergence or oscillation.

In all cases the forward/backward refinement generates shorter finish times than that of one-step scheduling.

In essence, the iterative forward-backward scheduling implicitly searches the space of possible schedules. The mechanism is not arbitrary and the search is somehow intelligent. As explained above, the procedure tries to establish an accurate estimate of tasks level values. In doing so, the graph is scanned up and down several times. As the iterative process proceeds, the mechanism filters out the most important tasks on the competing paths and investigates various assignments. The other secondary paths with less critical tasks play a less important role as the mechanism quickly evaluates their appropriate weight with respect to the graph and tends to stabilize their level values.

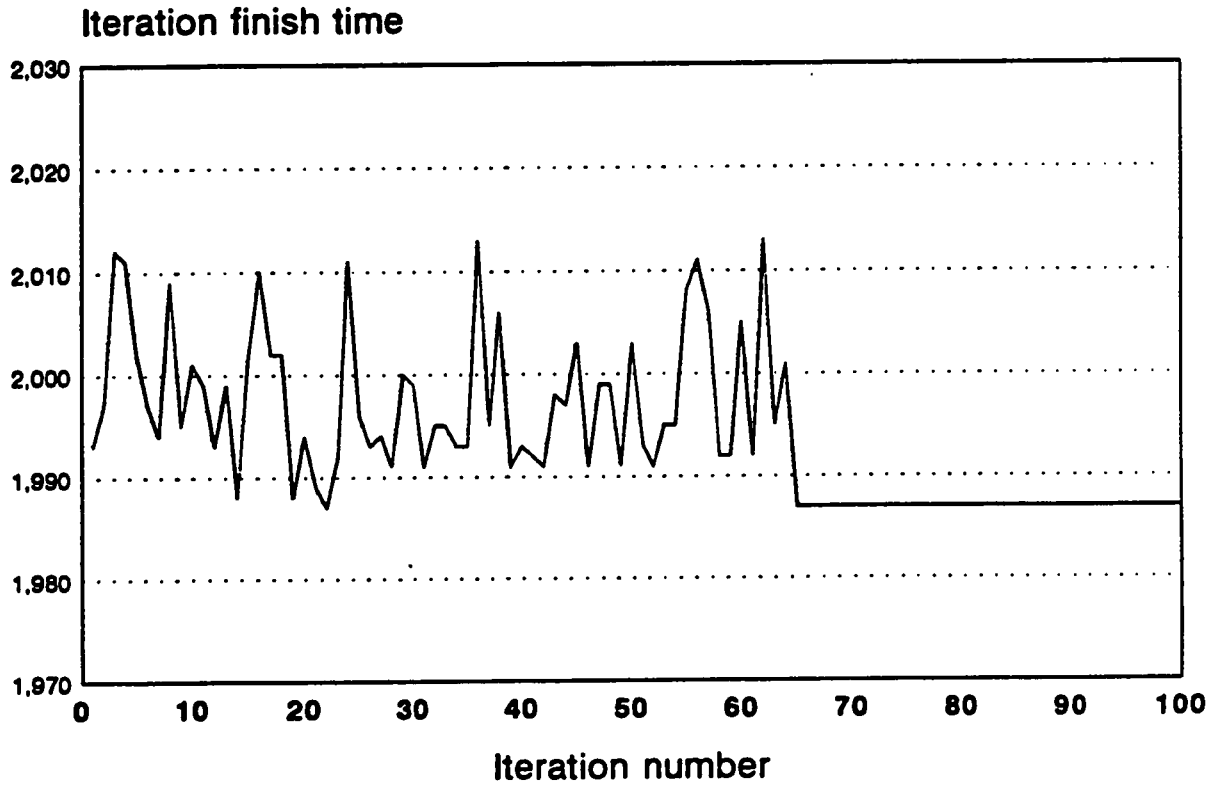

Figure 5.2: Iterative scheduling for the case where the schedule converges to the lowest value of all iteration.

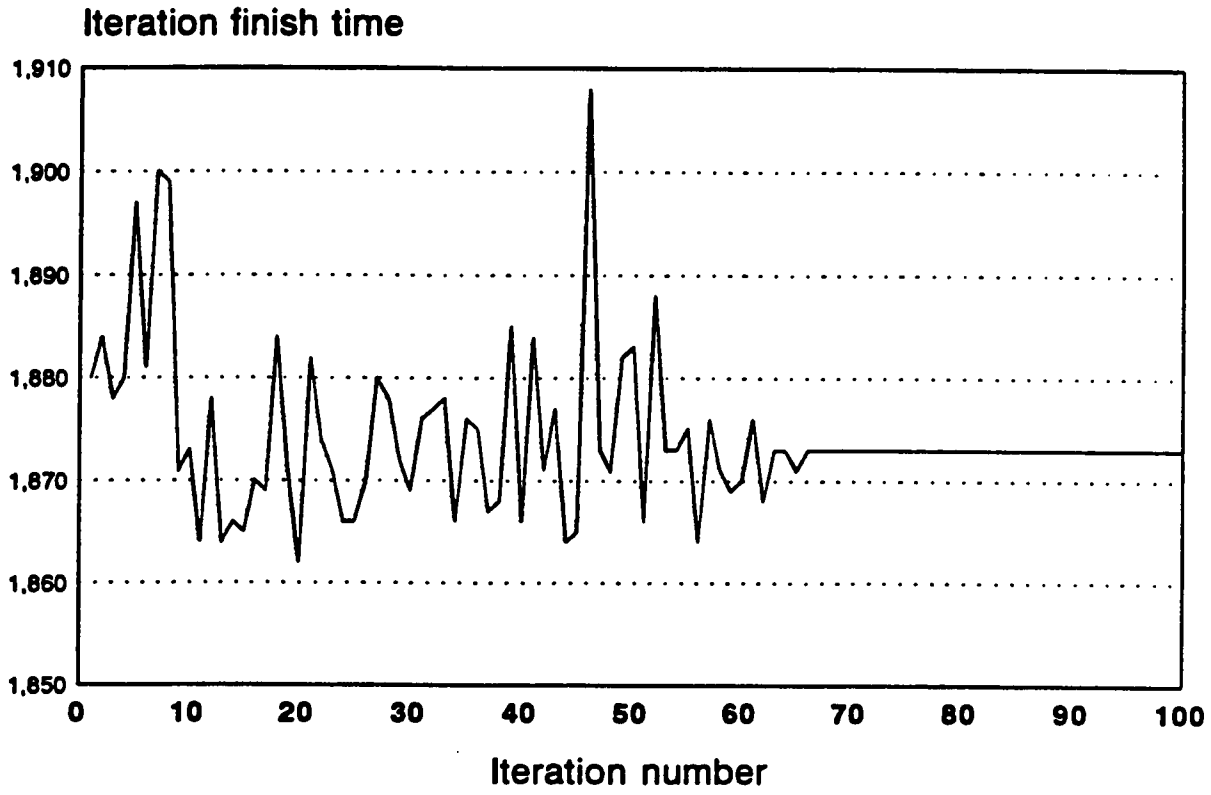

Figure 5.3: Iterative scheduling for the case where the schedule does not converge to the minimum iterations value.

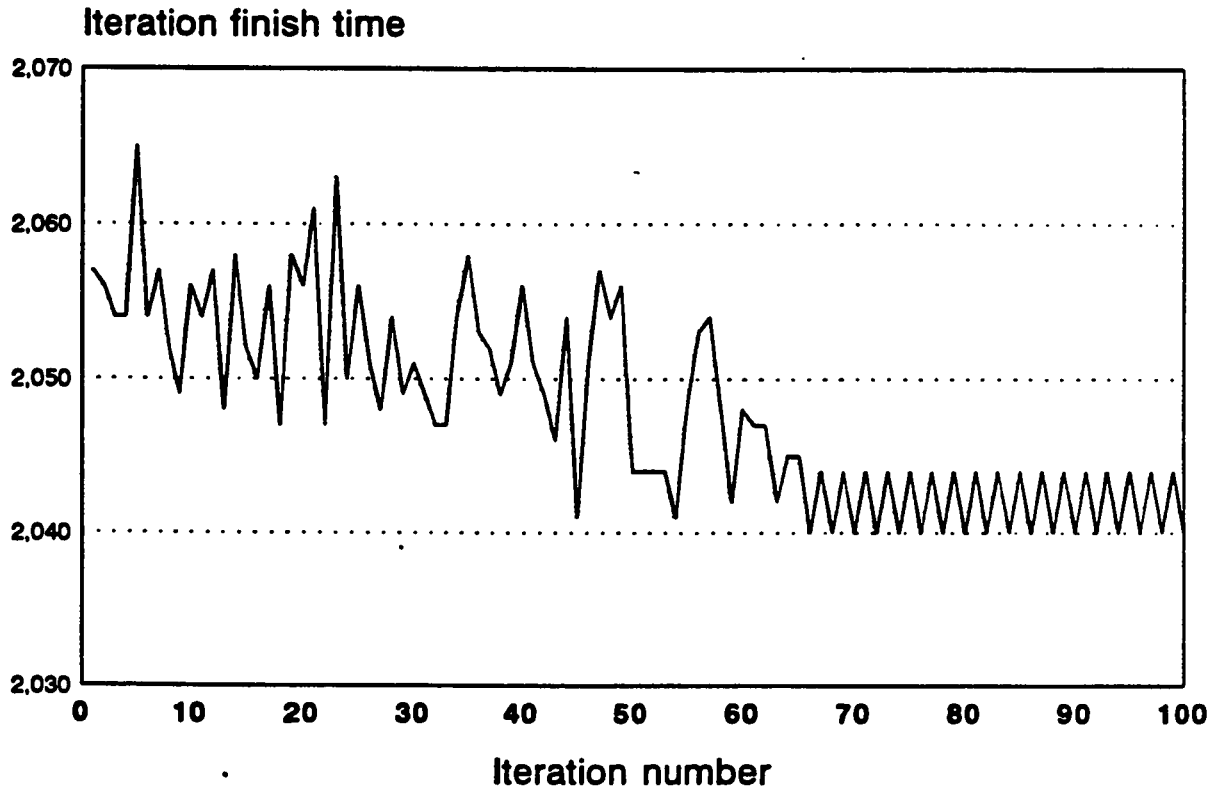

Figure 5.4: Iterative scheduling for the case where the process does not converge but oscillates.

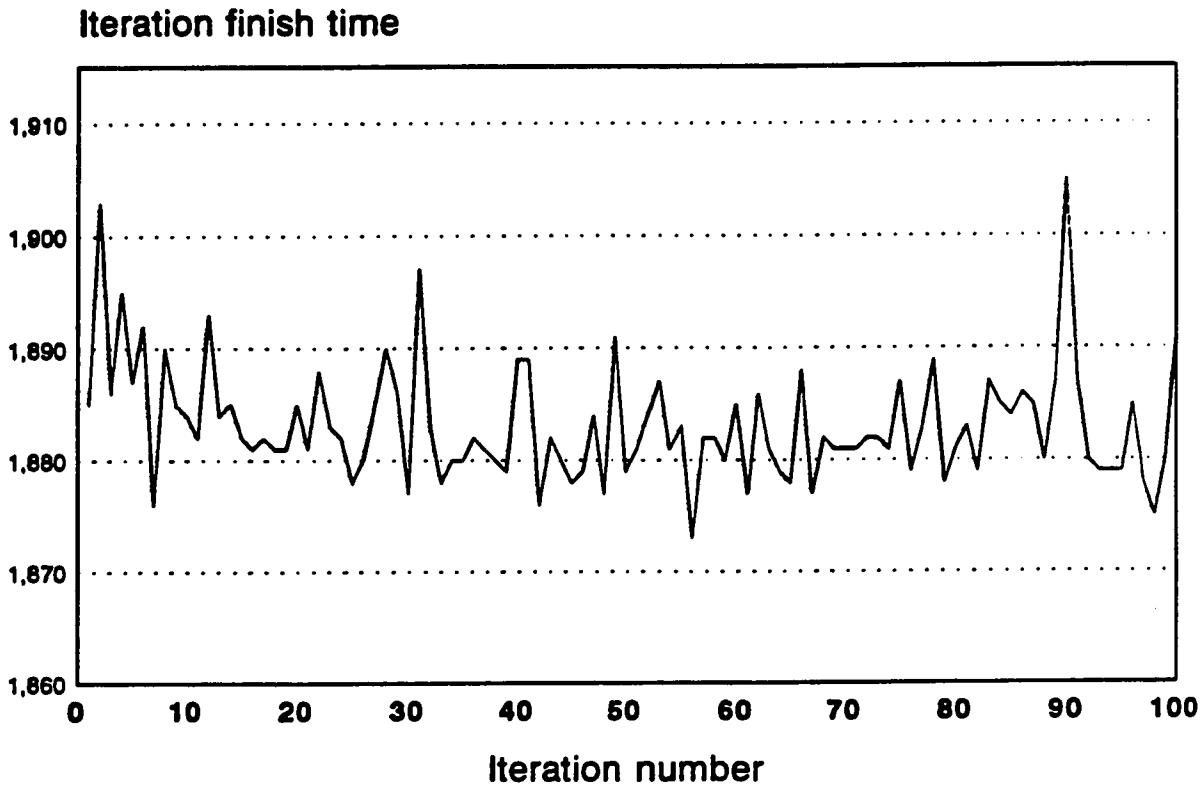

Figure 5.5: Iterative scheduling for the case where the process does not converge within 100 iterations.

5.5 Empirical Testing

The optimization technique was applied to some of the global heuristics presented in the previous chapter. The maximum number of iterations was limited to 100 iterations. However, for a given schedule, the process terminates if it reaches a stable finish time value for several consecutive iterations. In all cases, the corresponding number of iterations is recorded.

5.5.1 The effect of β and α

The number of iterations a heuristic goes through until it reaches a stable value[5] varies according to the values of β and α. It was found through testing that the number of iterations depends mainly on β and to a lesser extent on α (see figure 5.6). As shown in the figure, the number of iterations is increasing with β. β represents the degree of concurrency i.e the average number of tasks per processor in the set of ready to run tasks or the density of tasks with respect to resources. As β increases, the number of possible combinations of tasks on processors increases as well. This is so because, on the average, the number of tasks in the set of ready to run tasks is increasing with β. Consequently, the search space expands leading to an increase in the average number of iterations since there are more possible schedules to investigate. The increase of the number of iterations with α is due to the fact that increasing the communication means increasing the number of arcs and the arcs values which introduces more irregularity on the task graph and thus results in more unique and different possible tasks-assignments combinations. Figure 5.7 shows the case when the iterative technique is applied to a schedule characterized by low to medium values of β and α.

[5] A constant value. Oscillating values are considered as continuously changing

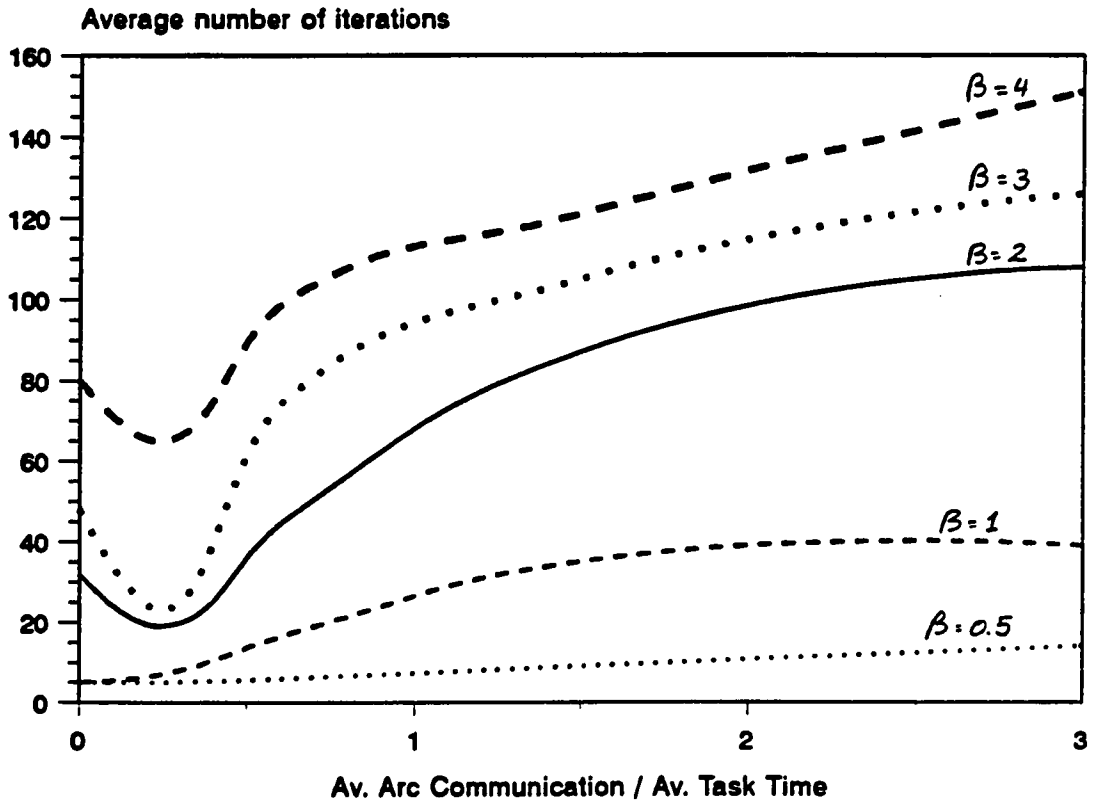

Figure 5.6: The effect of α and β on the number of iterations.

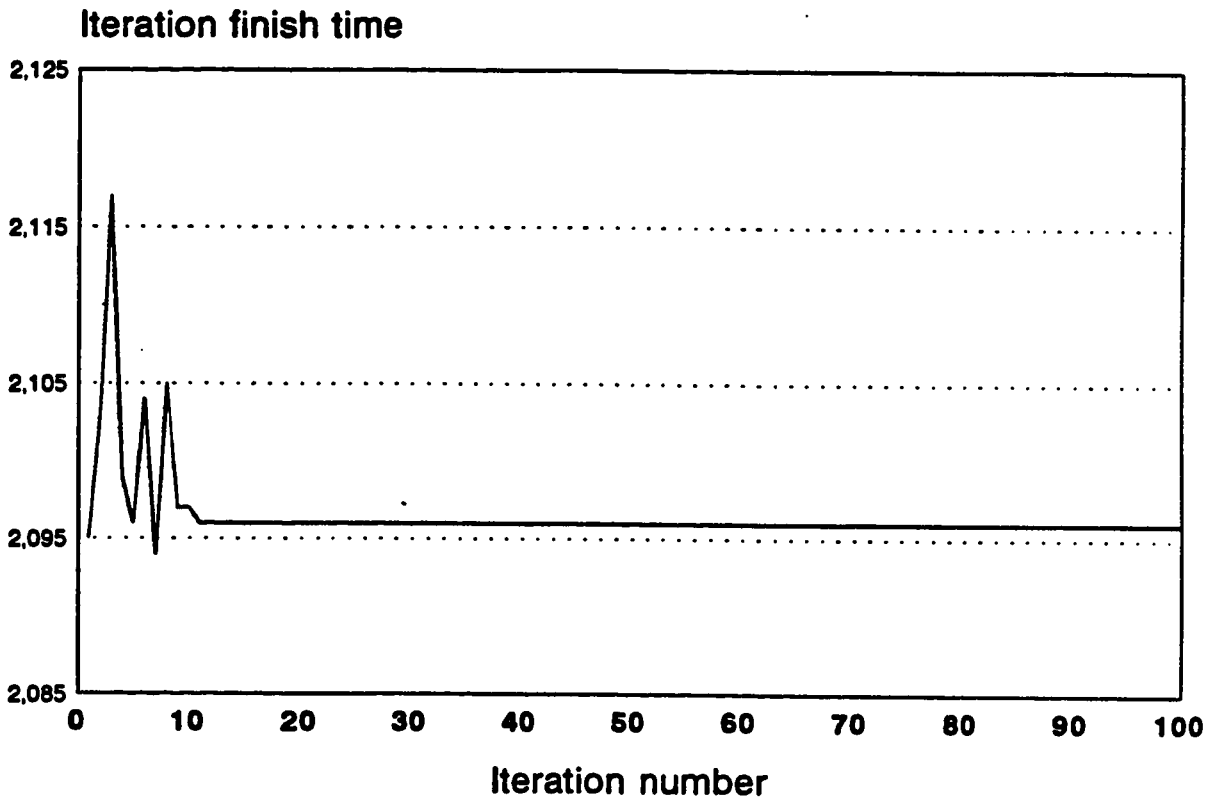

Figure 5.7: Low values of α and β lead to fast convergence.

5.5.2 Process Behavior

Most of the iteration graphs are characterized by subsequent high-low schedule finish time values. When testing relatively small graphs to study the effect of the iterative method within few iterations, the overall trend of the process led to shorter schedules. Figure 5.8 shows an example of a schedule that converges after some iterations. The tendency can be clearly seen in figure 5.9 where the logarithmic regression of figure 5.8 is drawn. Figure 5.9 is a typical example of the iterative process behavior. In general, the mechanism quickly finds better schedule finish times in the first few iterations. The process then slows down as finding better solutions becomes harder.

5.5.3 Analysis of the results

As mentioned before, the iterative technique was applied to several global heuristics. It might be thought that the improvement brought by applying this mechanism is due to the arbitrary selection of possible tasks assignments order in each iteration i.e testing possible schedule combinations and finding better ones arbitrarily. To investigate this point, we used another iterative mechanism which tries out different possible schedules of the graph. This is done by randomly chosing a possible schedule at each iteration. The choice is arbitrary so that at each iteration, we assign tasks randomly at their earliest starting time. This is similar to the random heuristic presented in the previous chapter but the process is repeated for a specified number of iterations so that at each iteration we pick a new possible schedule. The process can be seen as a random walk through the search space of possible schedules.

Figure 5.10 shows a comparison of the iterative $GD/HLETF^*$ and iterative *Random* both allowed to run for the same number of iterations. The graph clearly shows the different results obtained in each case. The iterative technique was able

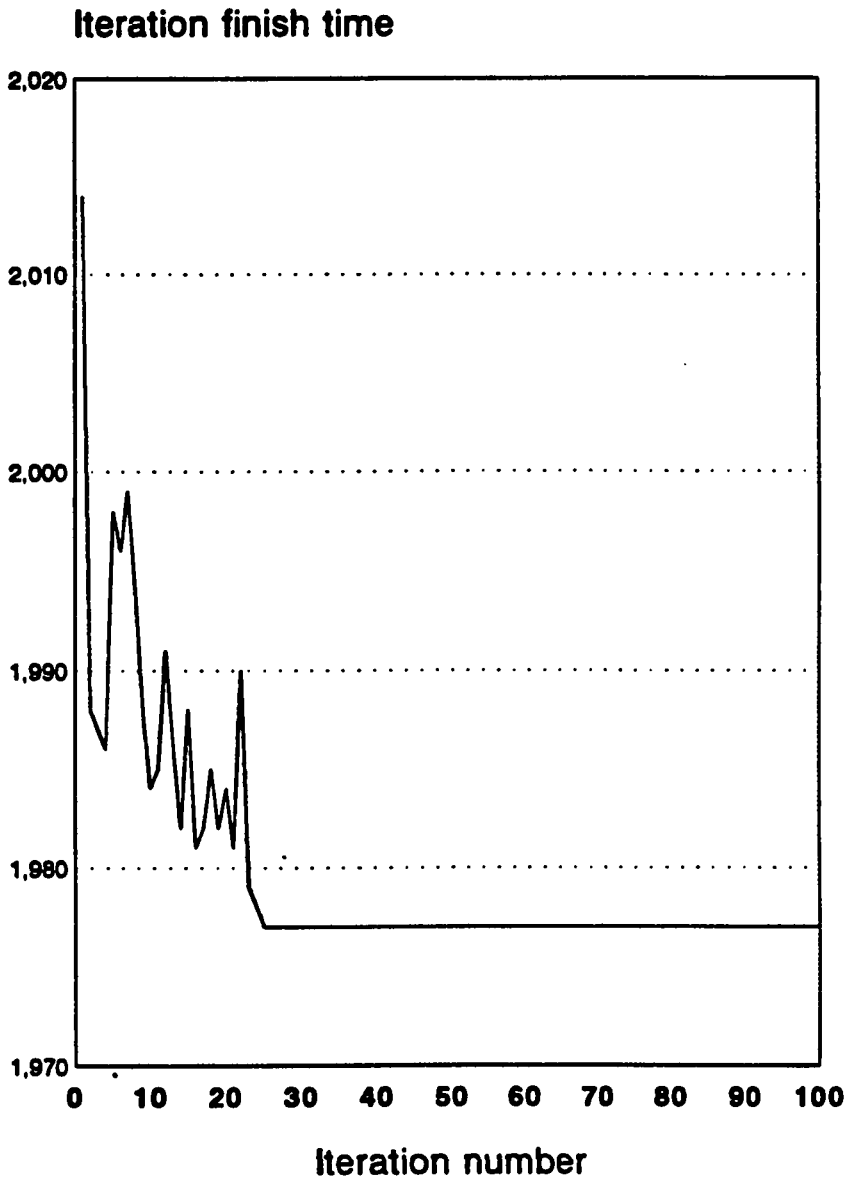

Figure 5.8: An example of iterative scheduling where the schedule converges to the lowest value.

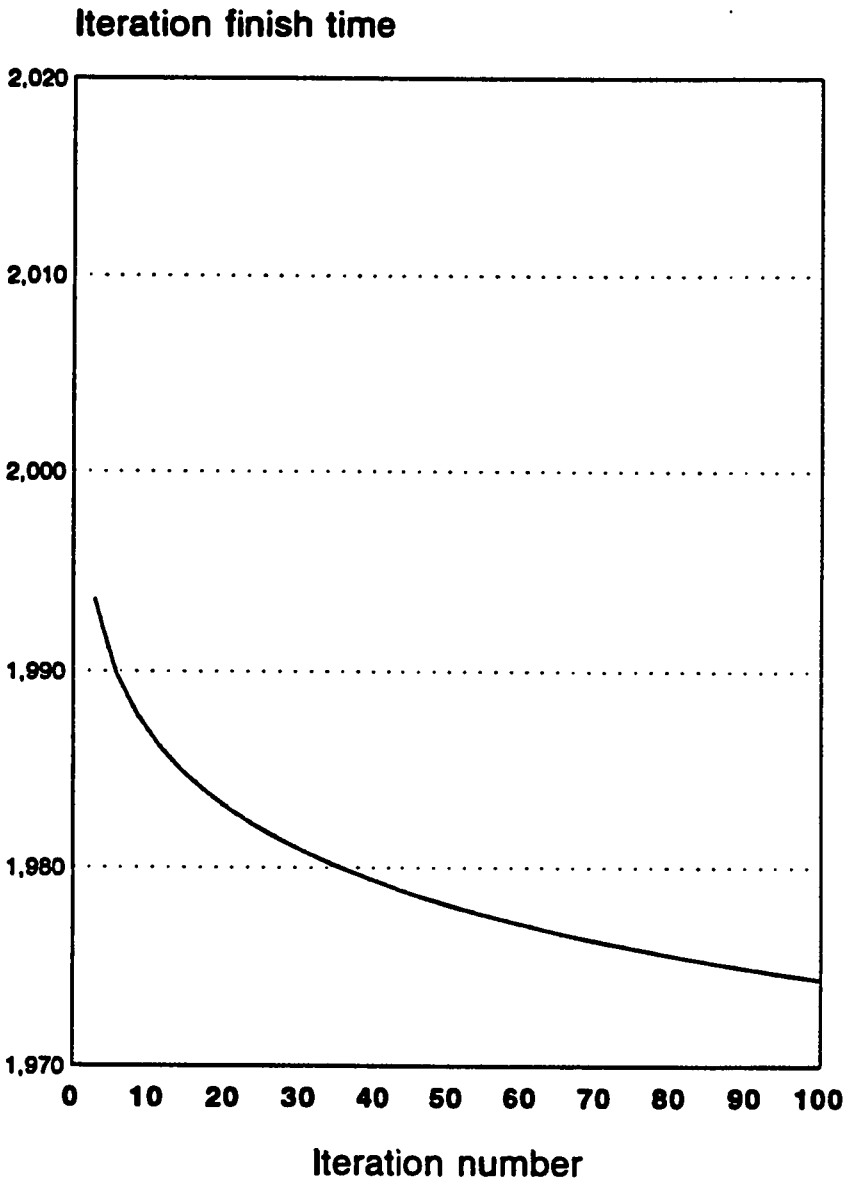

Figure 5.9: Logarithmic regression of the previous figure.

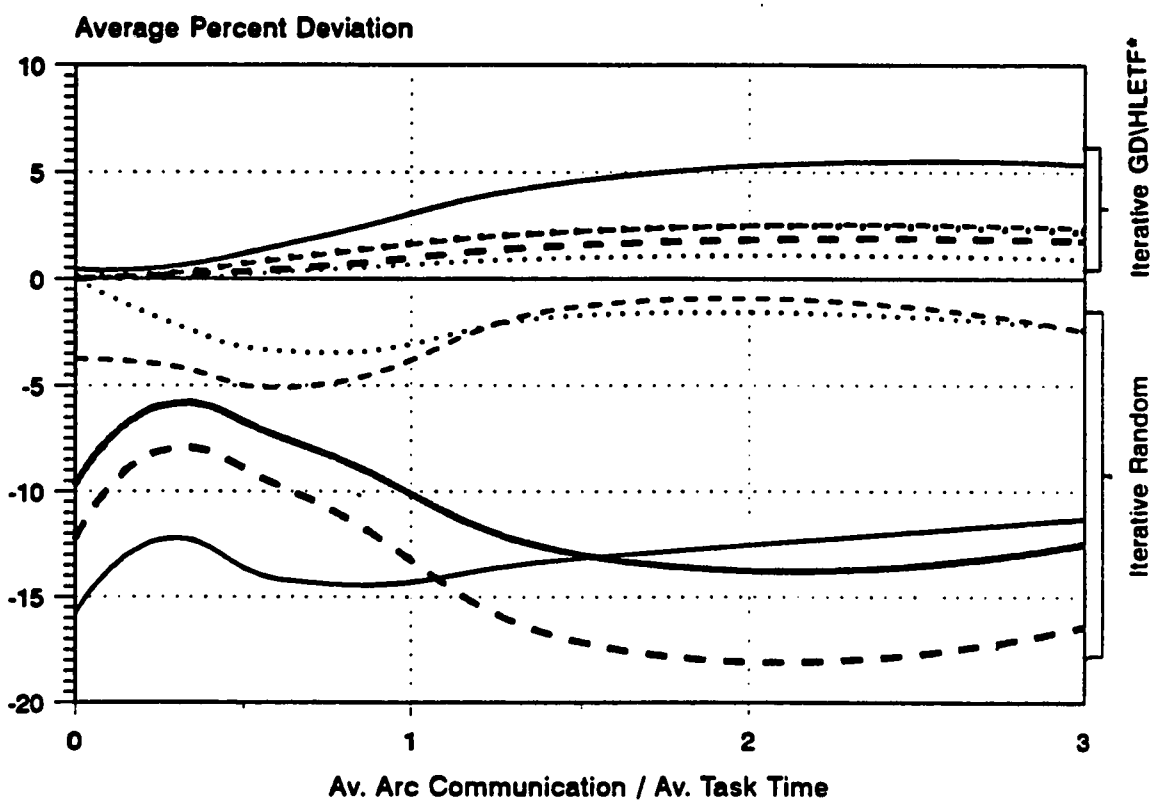

Figure 5.10: Iterative scheduling of $GD/HLETF^*$ heuristic vs. random walk.

to enhance our best heuristic ($GD/HLETF^*$) by 5% in some cases. Although iterative Random was able to produce better results than single-iteration Random (refer to the previous chapter), it was far from delivering a good schedule.

5.5.4 Random vs. Deterministic start

An interesting point to investigate in the optimization technique is to try and start the first iteration with random task level (*lst*) values. The optimization technique should be able then to quickly find realistic *lst* values in the first few iterations. Figure 5.11 shows a certain graph where the normal iterative $GD/HLETF^*$ (deterministic) is applied. The same heuristic is applied also with the optimization technique but with random initial *lst* tasks values. The graph shows that the random start was able to quickly recover from the inaccurate initial *lst* values and find the same finish time but with more iterations. An experiment was conducted using the random start for selected values of α and β and compared to the deterministic start. In general, when given enough iterations (100 iteration were allowed), the random start produced results only 1% (for $0.5 \leq \beta \leq 1$) to 2% (for $2 \leq \beta \leq 4$) less far from the results obtained using the deterministic start.

5.5.5 Improvements to Global Heuristics

Figure 5.12 shows the improvement obtained as a result of applying the iterative improvement technique to the GD/HLF heuristic for the fully connected topology. Figure 5.13 shows the results for the hypercube topology. Figure 5.14 and figure 5.15 show the improvement done to the $PD/HLETF$ heuristic for the two topologies. The iterative technique was successful in optimizing both graph-driven and processor-driven heuristics. The improvement is more in the case of the hypercube topology since the scheduling decision is more complex and can benefit more from the iterative process. Note that figure 5.10 showed the improvement to

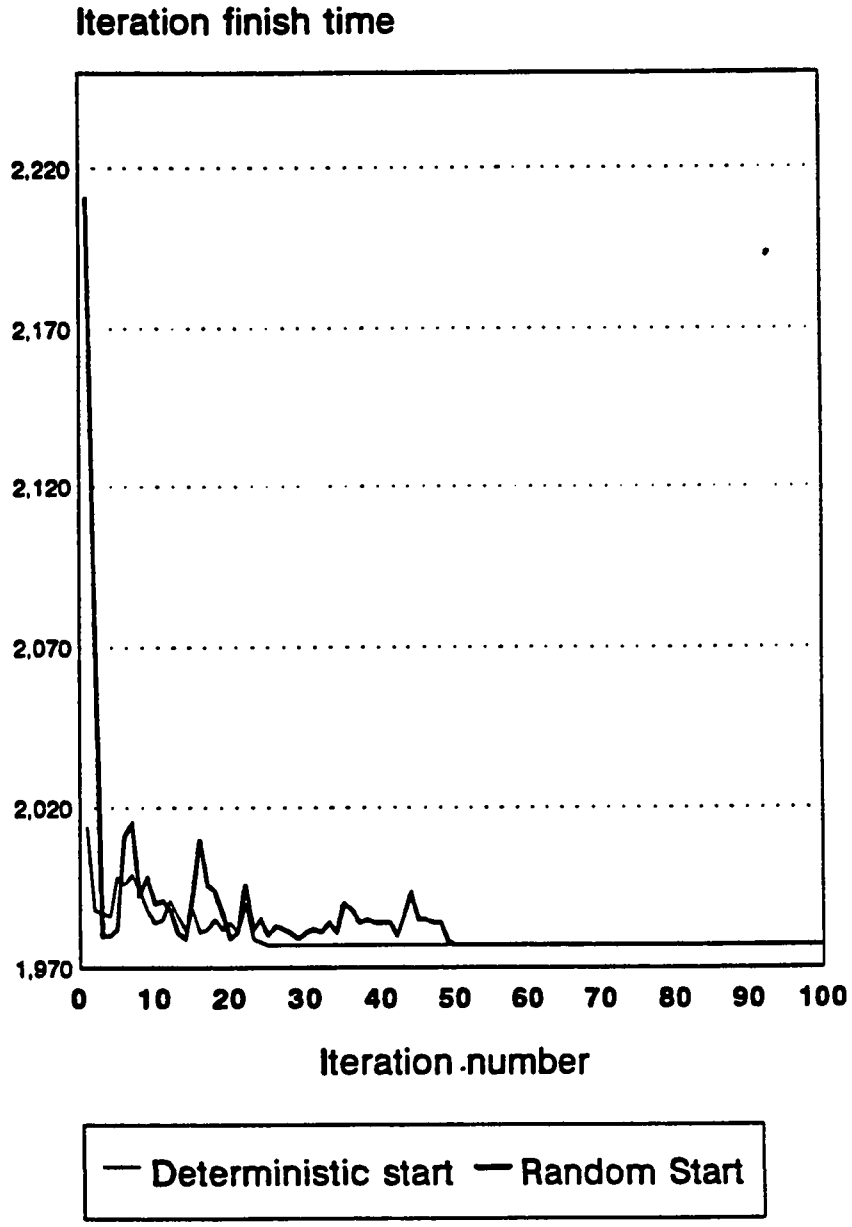

Figure 5.11: Comparison of Scheduling using Deterministic vs. Random start.

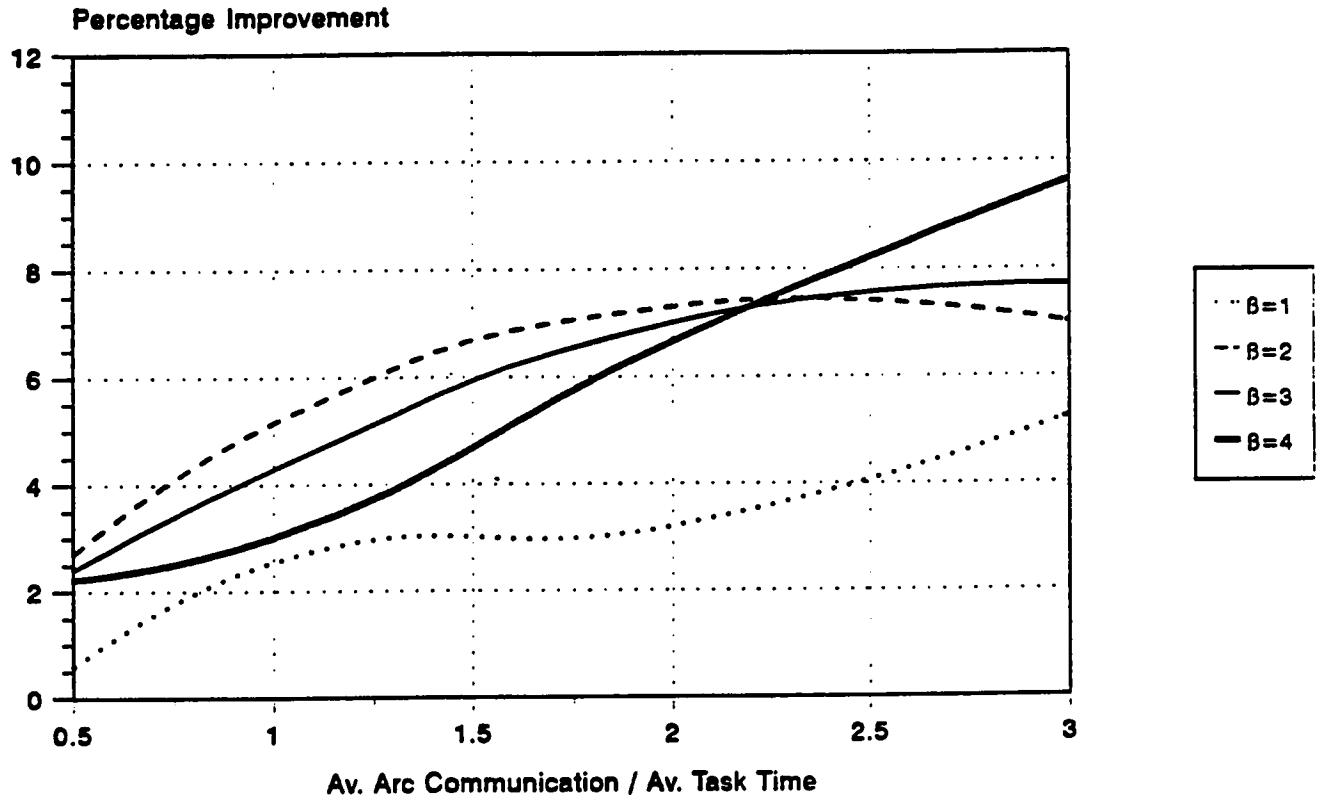

Figure 5.12: Percentage Improvement of iterative PD/HLETF over one iteration for FC topology.

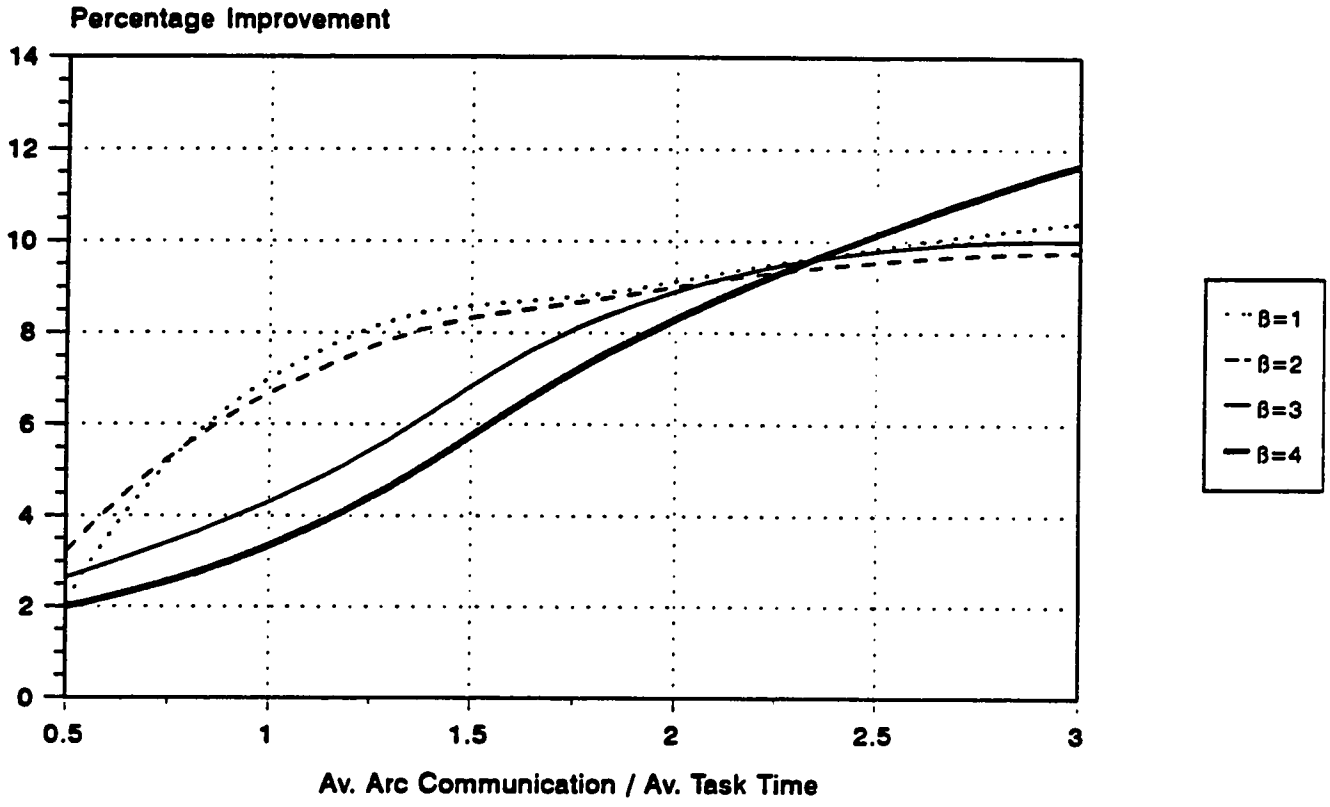

Figure 5.13: Percentage Improvement of iterative PD/HLETF over one iteration for HC topology.

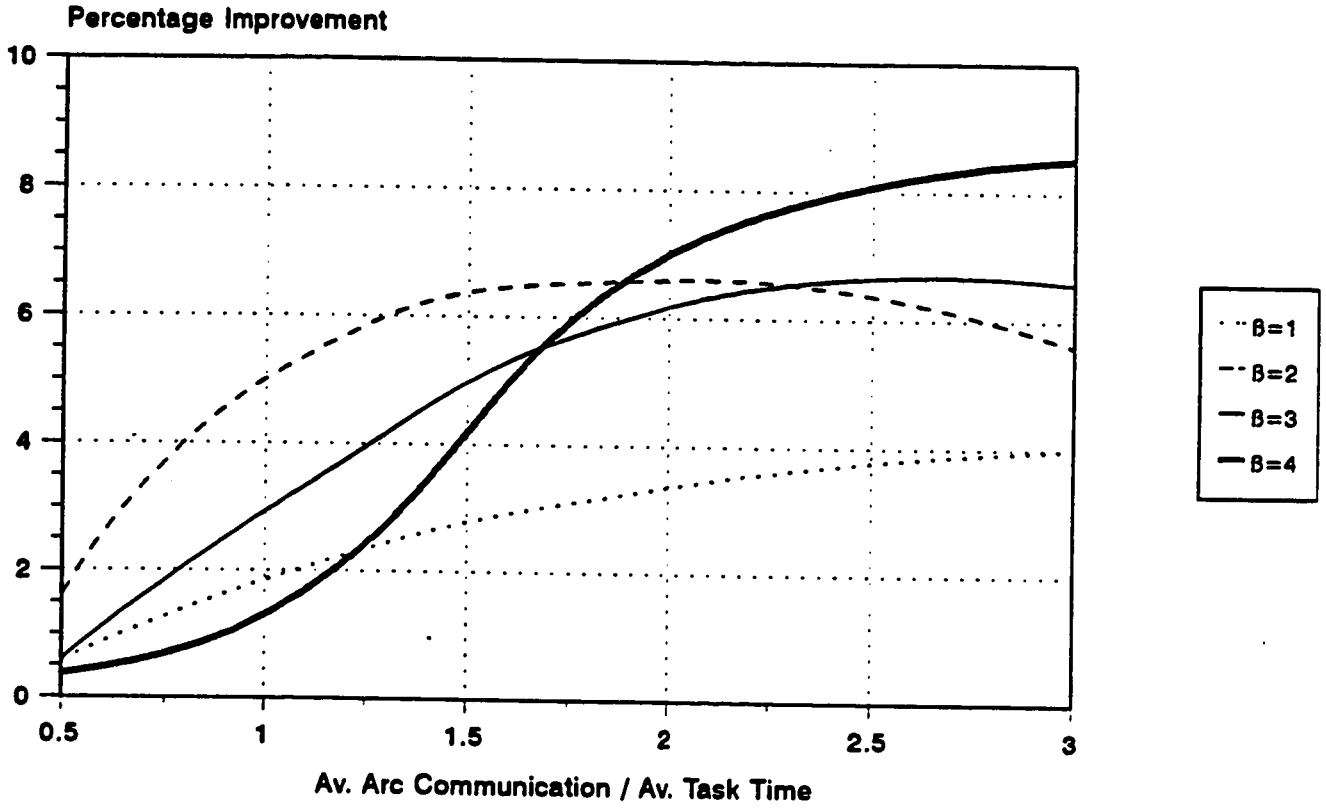

Figure 5.14: Percentage Improvement of iterative GD/HLF over one iteration for FC topology.

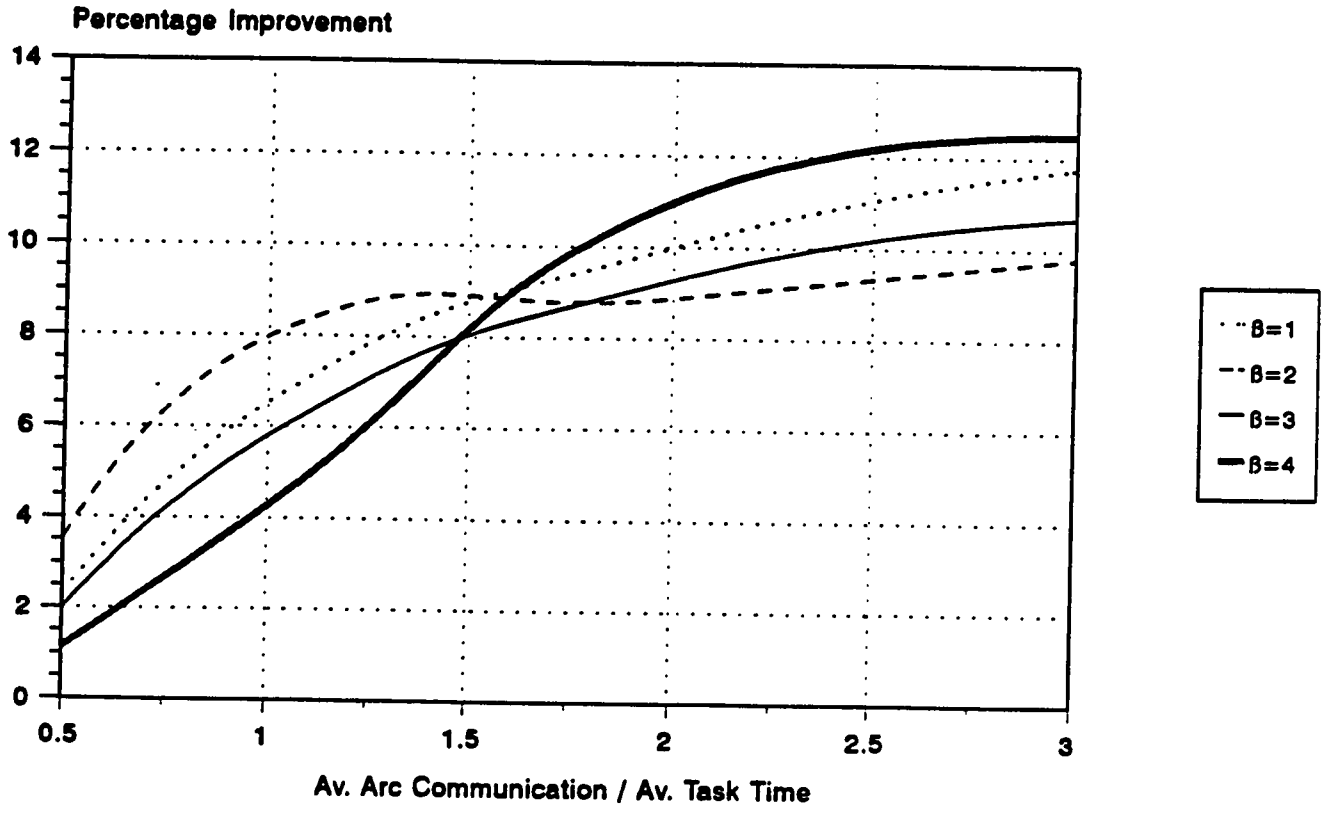

Figure 5.15: Percentage Improvement of iterative GD/HLF over one iteration for HC topology.

the $GD/HLETF^*$ heuristic.

The iterative improvement technique was also applied to the poor PD/HLF heuristic for some instances of α and β. For the case where $\alpha = 1.5$ and $\beta = 1$, 9.96% improvement was obtained. For the case where $\alpha = 1$ and $\beta = 2$, 16% improvement was obtained. For the case where $\alpha = 1.5$ and $\beta = 2.5$, 25.3% improvement was obtained. Finally for the case where $\alpha = 3$ and $\beta = 3$, 38% improvement was obtained. All improvements are with respect to the single iteration scheduling. The results (see figure 5.16) show that although the original heuristic had a poor time and priority management, the iterative improvement technique was still able to recognize critical tasks and give them higher and more accurate priorities.

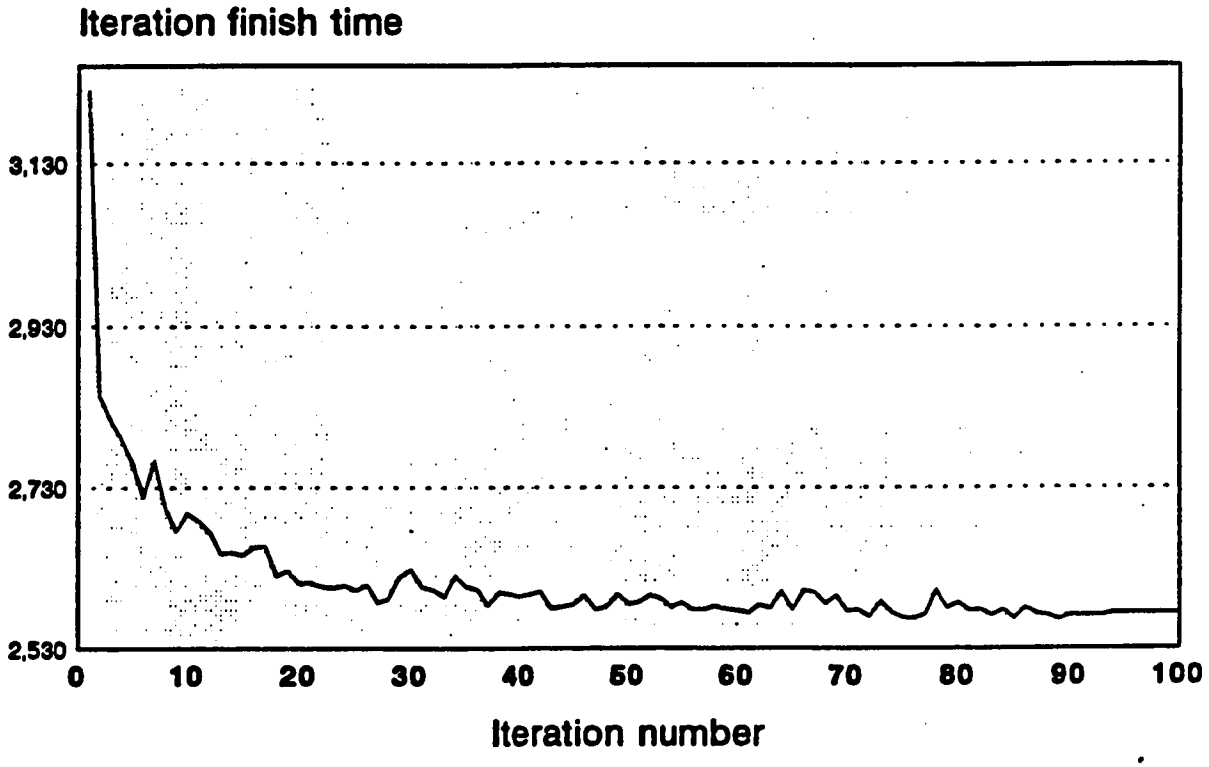

Figure 5.16: The rapid improvement to PD/HLF heuristic when applying the iterative technique.

Chapter 6

Conclusions and Future Work

In this work, an extensive survey of the scheduling algorithms for the models with and without communication was conducted. A new Generalized List Scheduling algorithm (GLS) for the model with communication was presented. This was done by implementing a new technique for evaluating the tasks *lst* values based on exploiting the characteristics of the reverse graph. The new scheduling algorithm then uses these values in defining the tasks priorities. The algorithm presented is graph driven as it was proved to be more suitable for handling global tasks priorities than the classical processor-driven approach. To evaluate the performance of the new heuristic, extensive empirical testing was conducted on several versions of (GLS) heuristics along with the best reported heuristic (ETF) for the model with communication. The results showed the performance advantage of GLS heuristics over ETF as the improvement ranged from 9% to 13%. Analysis of the results also showed the topology-independent-performance feature of GLS heuristic when applying different topologies.

The thesis also presented a new general iterative improvement technique for global scheduling heuristics. Extensive testing and analysis of the behavior of this technique was also conducted. Besides performing tests on the global heuristics

to show the improvements achieved over single iteration scheduling, several tests where also performed to study the behavior of this technique. Tests showed that even when starting with random *lst* values, the mechanism was able to quickly acquire the information about the characteristic of the graph and reflect that in delivering minimal schedules. For example, the tests conducted on the poor PD/HLF heuristic proved the ability of the iterative improvement technique to overcome the poor scheduling decision of the heuristic and produced much better results. The technique adapts itself not only to the graph structure but also to the heuristic used and produces for each type of heuristic the *lst* values that drive the scheduling decision to a shorter finish time. This is due to the nature of the process that senses the critical tasks which cause the schedule delay and give them the necessary priorities to schedule themselves earlier. Results obtained showed an average improvement of 3% for the best heuristic $GD/HLETF^*$ up to 30% worst-performing heuristics.

The following points summarize the cases where future problems and solutions can be investigated:

- Investigating a lower bound on the schedule finish time for the GLS heuristics.
- Generalizing the GLS heuristics to include the effect of contention. In doing so, a more realistic modeling of the scheduling process can be realized for practical systems. The problem complexity is expected to be higher specially when accounting for related issues of messages queuing and routing.
- Investigation of the cases where a convergence of the iterative improvement technique could be found. The investigation would study and state the necessary but sufficient conditions for the iterative process to converge on a finite number of iterations. Also, further study on the relation between the task graph characteristics and the number of iteration can be done.

- Application of GLS heuristics to stochastic models where the information regarding tasks execution times and communication messages can not be determined in advance.

Bibliography

[1] Coffman, E.G., and Denning, P.J., 'Operating Systems Theory' (eds) Prentice-Hall (1973).

[2] Coffman, E.G. et als, 'Computer and Job-Shop Scheduling Theory' (eds) John Willy & Sons (1976).

[3] Adam, T.L., Chandy, K.M., and Dickson, J.R., "A Comparison of list schedules for parallel processing systems", Comm. of the ACM, Vol 17, No 12, Dec. 1974, p.p. 685-690.

[4] Ramamoorthy, C.V. Chandy, K.M., and Gonzalez, M.J., "Optimal Scheduling Strategies in a Multiprocessor System" IEEE Trans. on Computers (Feb. 1972).

[5] Gonzalez Jr., M.J., "Deterministic Processor Scheduling", J. ACM Computing Surveys, Vol. 9, No. 3, Sep. 1977, pp. 173-204.

[6] Garey, M.R., Graham, R.L., and Johnson, D.S., "Performance guarantee for scheduling algorithms", Op. Research, Vol. 26, No 1, Jan. 1978, p.p. 3-21.

[7] Lo, V.M., "Heuristic Algorithm for Task Assignment in Distributed Systems", Proc. 4th Int. Conf. on Distri. Comput. Syst., May 1984.

[8] Lo, V.M., "Heuristic Algorithm for Task Assignment in Distributed Systems", IEEE Trans. on Computers, Vol. 37, No. 11, Nov. 1988, pp. 1384-1397.

[9] Cvetanovic, Z., "The effect of Problem Partitioning, Allocation, and Granularity on the Performance of Multiple-Processor Systems", IEEE Trans. on Computers, Vol. C-36, No.4, Ap.1987, pp.421-432.

[10] McGreary, C., and Gill, H., "Automatic Determination of Grain Size for Efficient Parallel Processing", Communications of the ACM, Vol.32, No. 9, Sep. 1989, pp. 1073-1078.

[11] Ashford Lee, E., and Bier, J., "Architectures for Statically Scheduled Dataflow", J. of Parallel and Distributed Computing, No. 10, 1990, pp.333-348.

[12] Pase, D.M., "A Comparative Analysis of Static Parallel Schedulers where Communication Costs are Significant", Ph.D. thesis, Oregon Graduate Center, Oregon, Jul. 1989.

[13] Hwang, J.-J., Chow, Y.-C., Anger, F.D., and Lee, C.-Y., "Scheduling precedence graphs in systems with interprocessor communication times", SIAM J. Computing, Apr. 1989, pp. 244-257.

[14] El-Rewini, H., and Lewis, T.G., "Scheduling Parallel Program Tasks onto Arbitrary Target Machines", J. of Parallel and Distributed Computing, No. 9, 1990, pp.138-153.

[15] Shirazi, B., Wang, M., and Pathak, G., "Analysis and evaluation of Heuristic Methods for Static Scheduling", J. of Parallel and Distributed Computing, No. 10, 1990, pp.222-232.

[16] Kasahara, H., and Narita, S., "Practical multiprocessor scheduling algorithms for efficient parallel processing", IEEE Trans. on Comp., Vol C-33, N0 11, Nov. 1984, pp. 1023-1029.

[17] Al-Mouhamed, M.A., "A multiprocessor system for real-time robotics applications", J. Microprocessors and Microsystems, Vol.X, No. Y, June, 1990, pp. 332.

[18] Fernandez, E.B., and Bussell, B., "Bounds on the number of processors and time for multiprocessors optimal schedules", IEEE Trans. on Comp., C-22, No 8, Aug. 1973, pp. 745-751.

[19] Al-Mouhamed, M., "Lower Bounds on the Number of Processors and Time for Scheduling Precedence Graphs with Communication Costs", IEEE Trans. on Software Engineering, Vol.16, No. 12, Dec. 1990, pp.1390-1401.

[20] Chen, C.L., Lee, C.S., Hou, E.S.H., "Efficient Scheduling Algorithm for Robot Inverse Dynamics Computation on a Multiprocessor System", IEEE Trans. on Sys., Man, and Cyber., Vol. 18, No. 5, Sep. 1988, pp.729-742.

[21] Prastien, M. Precedence-constrained scheduling with minimum time and communication. MS. thesis, University of Illinois at Urbana-Champaign, 1987.

[22] Sheild, J., "Partitioning concurrent VLSI simulated programs onto multiprocessor by simulated annealing", IEEE proceedings, No. 134, Jan. 1987, pp. 24-30.

[23] Lawler, E. L. and Wood, D. E., "Branch-and-Bound methods: A survey," Oper. Res., vol. 14, pp. 699-719, July 1966.

[24] Ibaraki, T., Combinatorial Optimization. Tokyo: Sangyo Tosyo, 1979.

[25] Kim, S.J. and Browne, J.C., "A general approach to mapping of parallel computation upon multiprocessor architecture", Proceedings of the Inter. Conf. on Parallel Processing, Vol.3, Aug. 1988, pp.1-8.

[26] Sarkar, V., and Hennessy, J., "Compile-time partitioning and scheduling of parallel programs", Proceedings of the SIGPLAN Symposium on Compiler Construction, Jul. 1986, pp. 17-26.

[27] Stone, H.S., "Multiprocessor scheduling with the aid of network flow algorithms", IEEE Trans. Softw. Eng. SE-3, pp. 88-93.

[28] Tanenbaum, A.S., and Renesse, R.V., "Distributed Operating Systems" Surveys, Vol. 17, No. 4, Dec. 1985.

[29] Anger, F.D., Hwang, J.J. and Chow, Y.C., "Scheduling with Sufficient Loosely Coupled Processors", Journal of parallel and distributed computing Vol.9 1990. pp. 87-92.

Vita

Adel Mohammed Adel Al-Massarani

Born at Riyadh, Saudi Arabia.

Received Bachelor's degree in Computer Engineering from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia June, 1990.

Completed Master's degree requirements at King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia in August 1993.